TECHNISCHE UNIVERSITÄT MÜNCHEN

TUM School of Computation, Information and Technology

Analyzing and Improving the Security of Trusted Execution Environments

Mathias Morbitzer

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:		Prof. Dr.-Ing. Jörg Ott
Prüfer*innen der Dissertation:	1.	Prof. Dr. Claudia Eckert
	2.	Prof. Dr. Uwe Baumgarten

Die Dissertation wurde am 03.04.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 26.09.2023 angenommen.

Acknowledgements

First of all, I would like to thank my thesis supervisor Prof. Dr. Claudia Eckert for giving me the possibility to pursue a PhD and Prof. Dr. Uwe Baumgarten for being the second examiner of this thesis. Also, I would like to thank Dr. Julian Schütte for his guidance especially in the initial phase of this journey as well as Christian Banse and Gerd Brost for making sure that I do not get buried by other projects.

Furthermore, I would like to thank my colleagues at Fraunhofer AISEC for the fruitful discussions and their valuable feedback. With the guilty feeling of forgetting someone, this includes Georg Bramm and Thomas Bellebaum for their input, everyone I ever discussed my work with during a coffee break or a meeting, and especially Dr. Martin Schanzenbach for his detailed feedback on earlier versions of this work.

Additionally, I would like to thank my coauthors, in alphabetical order: Dr. Robert Buhren, Marko Dorfhuber, Prof. Dr.-Ing. Thomas Eisenbarth, Dr. Felicitas Hetzelt, Dr. Julian Horsch, Dr. Manuel Huber, Benedikt Kopf, Dr. Sergej Proskurin, Erick Quintanar Salas, Martin Radev, Jan Wichelmann, Luca Wilke, and Philipp Zieris.

Last but not least, I want to express my sincere gratitude to my parents for their constant backing over all these years and to Laure for taking this journey with me and her encouragement, support and patience along the way.

Abstract

Modern enterprises increasingly shift potentially critical applications into cloud environments. Yet, by outsourcing sensitive code and data into the cloud, enterprises are required to trust the cloud provider not to meddle with the processed data. To address such concerns when moving critical tasks into the cloud, the majority of cloud providers offers users the possibility to deploy Trusted Execution Environments (TEEs). TEEs are a combination of software and hardware mechanisms which aim to protect their data and processes from high privileged adversaries. To achieve this goal, TEEs deploy different protection mechanisms. In this thesis, we investigate the two protection mechanisms implemented by all TEEs provided by the industry: memory isolation and attestation.

We investigate the memory isolation mechanism by analyzing the impact of missing memory integrity protection on the isolation, using the example of AMD Secure Encrypted Virtualization (SEV). SEV aims to achieve its memory isolation by encrypting the memory of Virtual Machines (VMs) with a unique key. This key is maintained in hardware, rendering the encrypted memory inaccessible to the cloud provider. Yet, SEV refrains from protecting the integrity of the encrypted memory. We show how this missing memory integrity protection can enable a malicious administrator to utilize a service running within the VM to provide arbitrary decrypted memory content, allowing to extract secrets from the protected VM within seconds. By describing our attack in detail, we determine the attack vectors required for a successful exploitation. Among these attack vectors, we identify the missing Second Level Address Translation integrity protection to be the most critical. Furthermore, we show that this can be combined with other attack vectors to affect not only the VM's confidentiality, but also its integrity. We achieve this goal by remapping one of the VM's kernel functions to a payload which we inject into the VM's memory by sending it a network packet. This permits us to execute arbitrary code within SEV-protected VMs.

Having showed how the attacks can be performed, we iterate through the determined attack vectors to discuss how SEV's memory isolation could be restored. Most importantly, we propose a change to SEV which prevents a targeted modification of the Second Level Address Translation by a malicious hypervisor. Our solution only requires a small modification to how SEV encrypts the memory of protected VMs. However, it permits to efficiently prevent a high privileged adversary from performing meaningful modifications of the VM's memory by including the guest physical address of the encrypted data into the encryption.

Afterwards, we discuss improving the attestation mechanism of TEEs by extending the attestation of the initial state with attestation at run-time. Specifically, we present an architecture that enables a monitor in a TEE to collect run-time information of an application running in a different TEE. Based on this architecture, we establish a design that allows us to use control-flow attestation to ensure the integrity of the application. By placing all components of our design in TEEs, we are able to not only detect a compromised application, but also protect ourselves from malicious administrators. We show the practicability of our design by providing a prototype based on Intel Software Guard Extensions (SGX), with which we perform a detailed performance evaluation in Microsoft Azure. Our evaluation shows that the need to transfer information between TEEs and the additional verification process add considerable overhead. Yet, we are able to reduce this overhead by securely caching collected information and by performing the analysis in parallel to executing the application. This shows that our design for control-flow attestation in TEEs provides a practical solution for cloud users focused on protecting the integrity of their data and processes at run-time.

Zusammenfassung

Immer mehr Firmen verlagern kritische Prozesse in die Cloud. Durch die Auslagerung von kritischen Programmen und Daten sind die Firmen gezwungen, darauf zu vertrauen, dass der Cloud-Anbieter diese nicht beeinflusst. Um diese Sorgen bei der Verlagerung von kritischen Daten in Cloud-Umgebungen zu adressieren, bietet der Großteil der Cloud-Anbieter die Möglichkeit zur Verwendung von Trusted Execution Environments (TEEs). Bei TEEs handelt es sich um eine Kombination von Software- und Hardwaremechanismen, welche das Ziel haben, Daten innerhalb der Umgebung vor Angreifern mit hohen Privilegien zu schützen. Um dieses Ziel zu erreichen, setzen TEEs auf verschiedene Schutzmechanismen. In dieser Arbeit beschäftigen wir uns mit zwei Schutzmechanismen, die in allen von der Industrie angebotenen TEEs zum Einsatz kommen: Speicherisolierung und Attestierung.

Für die Untersuchung der Speicherisolierung analysieren wir die Auswirkungen von fehlender Integrität des Arbeitsspeichers auf die Isolierung anhand des Beispiels von AMD Secure Encrypted Virtualization (SEV). SEV versucht die Speicherisolierung durch die Verschlüsselung des Arbeitsspeichers von virtuellen Maschinen (VMs) mit einem separaten Schlüssel zu erreichen. Der hierfür notwendige Schlüssel wird in Hardware verwaltet, was den verschlüsselten Arbeitsspeicher unzugänglich für den Cloud-Anbieter macht. Jedoch verzichtet SEV darauf, auch die Integrität des verschlüsselten Arbeitsspeichers sicherzustellen.

Wir zeigen dass diese fehlende Integrität es einem Angreifer ermöglicht, einen Service innerhalb der VM zur Extraktion von beliebigem Inhalt des verschlüsselten Arbeitsspeichers im Klartext zu verwenden. Dadurch ist es möglich, innerhalb von Sekunden kritische Daten aus der geschützten VM zu extrahieren. Anhand der detaillierten Beschreibung des Angriffs ermitteln wir, welche Angriffsvektoren für einen erfolgreichen Angriff notwendig sind. Unter diesen Angriffsvektoren identifizieren wir den fehlenden Integritätsschutz der Secondary Level Address Translation als kritischsten Punkt. Des Weiteren zeigen wir, dass dieser in Kombination mit anderen Angriffsvektoren auch verwendet werden kann, um nicht nur die Vertraulichkeit der VM, sondern auch ihre Integrität zu brechen. Wir erreichen dieses Ziel, indem wir die Adresse einer Kernelfunktion innerhalb der VM einem Payload zuordnen, welchen wir mithilfe eines Netzwerkpakets in die VM einschleusen. Dies erlaubt es uns, beliebigen Code innerhalb der mit SEV geschützten VM auszuführen.

Nach der detaillierten Darstellung der Angriffe analysieren wir die einzelnen Angriffsvektoren und besprechen wie diese behoben werden könnten, um SEV's Speicherisolierung wiederherzustellen. Unser wichtigster Beitrag zu dieser Diskussion ist eine Modifikation von SEV, welche gezielte Veränderungen der Anordnung des Arbeitsspeichers durch einen böswilligen Hypervisor verhindert. Unsere Gegenmaßnahme besteht aus einer kleinen Modifikation des von SEV verwendeten Verschlüsselungsalgorithmus mit einer großen Wirkung. Diese kleine Modifikation verhindert die gezielte Veränderung der Anordnung des Arbeitsspeichers durch einen Angreifer mit hohen Privilegien durch die Einbindung der physikalischen Adresse des Gastes in die Verschlüsselung.

Danach beschäftigen wir uns mit der Verbesserung des Attestierungsmechanismus von TEEs durch die Erweiterung der Attestierung des initialen Status der TEE auf eine Laufzeitattestierung. Im Detail stellen wir eine Architektur vor, welche es einem Prozess in einer TEE erlaubt, Laufzeitinformationen über eine Applikation in einer anderen TEE zu sammeln. Basierend auf dieser Architektur entwickeln wir ein Design, welches es uns ermöglicht, Kontrollflussattestierung zu verwenden, um die Integrität einer Applikation sicherzustellen. Die Platzierung aller Komponenten des Designs in TEEs erlaubt uns nicht nur die Erkennung von Angriffen auf die Applikation, sondern zusätzlich den Schutz vor Angreifern mit hohen Privilegien. Wir zeigen die Anwendbarkeit unseres Designs durch die Bereitstellung eines Prototypen basierend auf Intel Software Guard Extensions (SGX), welchen wir detailliert in Microsoft Azure evaluieren. Unsere Evaluierung zeigt, dass die Notwendigkeit, Informationen zwischen zwei TEEs auszutauschen, sowie die zusätzliche Verifikation beträchtliche Kosten verursachen. Es ist uns jedoch möglich, diese Kosten durch die sichere Zwischenspeicherung von gesammelten Informationen sowie die Parallelisierung der Analyse und der Ausführung der Applikation zu reduzieren. Dies zeigt, dass unser Design für Kontrollflussattestierung in TEEs eine praktische Lösung für Cloud-Benutzer darstellt, welche die Integrität ihrer Daten und Prozesse zur Laufzeit sicherstellen wollen.

Contents

Acknowledgements i

Abstract iii

Zusammenfassung v

1 Introduction 1
- 1.1 Research questions . . . 4
- 1.2 Contributions . . . 6
  - 1.2.1 Contributions to RQ1 . . . 6
  - 1.2.2 Contributions to RQ2 . . . 7
  - 1.2.3 Contributions to RQ3 . . . 7
- 1.3 Publications . . . 8
- 1.4 Outline . . . 10

2 Background 11
- 2.1 AMD SEV . . . 11
- 2.2 Intel SGX . . . 16
- 2.3 Control-Flow Attestation . . . 18

3 State of the Art 21
- 3.1 Attacks Against AMD SEV . . . 21
- 3.2 Defenses for AMD SEV . . . 23
- 3.3 Memory Integrity Monitoring . . . 25
- 3.4 Summary . . . 27

4 Analyzing the Impact of Missing Memory Integrity by Example 29
- 4.1 Attacker Model . . . 30
- 4.2 Impact of Missing Memory Integrity on the Confidentiality of TEEs . . . 31
  - 4.2.1 VM Page Access Tracking . . . 32
  - 4.2.2 Resource Identification . . . 33
  - 4.2.3 Secret Identification . . . 39
  - 4.2.4 Secret Extraction . . . 40
  - 4.2.5 Evaluation . . . 41

4.2.6 Discussion . . . 51
4.3 Impact of Missing Memory Integrity on the Integrity of TEEs . . . 53
4.3.1 Trigger Identification . . . 55
4.3.2 Payload Destination Identification . . . 57
4.3.3 Payload Execution . . . 59
4.3.4 Evaluation . . . 60
4.3.5 Discussion . . . 63
4.4 Summary . . . 65

5 Restoring Cryptographic Memory Isolation without Memory Integrity 69
5.1 Countermeasures by the VM . . . 69
5.2 Countermeasures by SEV . . . 71
5.3 Summary . . . 74

6 Using TEEs to Verify Run-Time Integrity 77
6.1 Attacker Model . . . 78
6.2 Enabling TEEs to Measure Run-Time Integrity of Applications . . . 78
6.3 Porting Control-Flow Attestation to TEEs . . . 81
6.3.1 Overview . . . 82
6.3.2 Offline Phase . . . 83
6.3.3 Online Phase . . . 86
6.4 Implementation . . . 86
6.4.1 Overview . . . 87
6.4.2 Instrumentation . . . 89
6.4.3 Execution . . . 92
6.5 Evaluation . . . 94
6.5.1 Setup . . . 94
6.5.2 Benchmark Performance Evaluation . . . 95
6.5.3 Component Performance Evaluation . . . 95
6.5.4 Signing Service Performance Evaluation . . . 97
6.6 Discussion . . . 99
6.7 Summary . . . 102

7 Conclusions 105
7.1 Contributions to Research Questions . . . 105
7.2 Future Work . . . 106

List of Figures 109

List of Tables 111

Acronyms 113

Glossary 117

Bibliography **119**

CHAPTER 1

Introduction

Cloud computing has become omnipresent in our everyday life. Although this is not always obvious to us, we get reminded of this fact when our vacuum cleaners and door bells refuse to work due to outages in cloud infrastructure [10]. Additionally to affecting the availability of our household items, cloud computing can also affect the confidentiality and integrity of our data, which is also often stored in the cloud. Incidents exposing this often sensitive data underline the risk of storing data in the cloud. To be precise, 98% of all companies using cloud infrastructure reported that they experienced at least one data breach within the last 18 months [46].

More than half of these breaches were related to insufficient access management [46]. An example is the Capital One breach, in which attackers gained access to more than 100 million customer accounts and credit card applications [99]. Experts claim that the breach could have been prevented by applying defense mechanisms such as the principle of least privilege [59]. This principle dictates that entities such as programs, systems and objects only receive the minimal number of privileges required to perform their tasks [134].

To achieve this goal, the entities can be assigned to different, hierarchically structured protection domains with ascending privileges. While the permissions of the least privileged domain are limited, the highest privileged domain possesses all privileges. An example for such hierarchical protection domains are the protection rings in the x86 architecture [67]. In this model, the least privileged ring, ring 3, is intended for user-space applications, which require only a limited amount of permissions. In comparison, the most privileged ring, ring 0, is meant to host the kernel, which requires higher privileges. To protect software running in higher privilege rings from lower privileged adversaries, processes are forbidden to access data from higher privilege rings. However, accessing data from lower privilege rings is permitted. This prevents user-space applications from interfering with the kernel, while at the same time giving the kernel the possibility to manage running applications.

Virtualization follows a similar approach [134]. Using virtualization, the hypervisor, running in a high privilege domain, hosts a Virtual Machine (VM) running in a lower privilege domain. By using different privilege domains, the hypervisor is protected from

a malicious VM. Yet, the hypervisor can still access all of the VM's data, enabling it to perform operations such as managing the VM's memory.

These examples show how hierarchical protection domains allow to protect software in higher privilege domains from adversaries in lower privilege domains. Yet, in various scenarios, the lower privileged software additionally may not trust higher privilege layers. A prime example for this limited trust into higher privilege layers is cloud computing. Specifically, a large number of enterprises have concerns about the limited ability to restrict and detect access to data hosted in cloud environments by, for example, a curious or malicious administrator [72]. While many may not deem their cloud provider to be malicious, we also need to consider the possibility of an attacker gaining elevated privileges or exploiting software vulnerabilities in the cloud provider's infrastructure. The past has shown that such breaches can happen in the cloud by escaping an isolation domain [103] or by exploiting bugs in services shared among different users [114].

Furthermore, the need to protect software from higher privilege domains not only applies to the cloud, but also to our personal devices. This requirement has been highlighted by the COVID-19 pandemic and the resulting shift to working from home. Most companies were not prepared for this sudden shift, leading to more than half of all employees using personal devices when working from home [110]. Yet, in the past, personal devices being used for business tasks caused attacks compromising data of tens of thousands of customers [152]. To avoid such attacks, companies have to ensure the confidentiality and integrity of work-related tasks despite not having administrative access to the system performing the tasks.

Hence, mechanisms which protect software from higher privilege layers are required both on personal devices and in cloud environments. For that reason, previous work proposed the concept of Trusted Execution Environments (TEEs) [138, 96, 27, 47, 35, 145, 87]. TEEs aim to protect data and processes within the TEE from adversaries, even if they are located at higher privilege layers. To be able to protect against such a strong threat model, most TEEs make use of a combination of hardware and software mechanisms. In recent years, the industry also started to provide different TEEs, making them accessible to a wider audience [17, 34, 75, 120, 69, 64, 8]. Seeing TEEs as a possibility to reduce their customer's worries about the limited control over data hosted in the cloud, also major cloud providers meanwhile support TEEs [126, 76, 48, 71, 121].

To ensure the protection of data and processes within TEEs, such environments can protect their memory either via logic or cryptographic isolation [129]. Logical isolation deploys access control mechanisms to prevent high privileged adversaries from accessing the TEE's memory. In comparison, cryptographic isolation does not prevent the adversaries from accessing the TEE's memory, but instead encrypts the memory with a key inaccessible to the adversary. Hence, even though being able to access the encrypted memory, the adversary cannot retrieve its plaintext due to the missing access to the cryptographic material required for decryption. This allows cryptographic isolation to prevent an adversary without access to the encryption key from reading or writing meaningful data from or into the TEE's memory.

Due to these safety properties, some TEEs consider memory confidentiality sufficient to isolate their memory from high privileged adversaries, and refrain from also protecting the memory's integrity. An example for such a TEE is AMD Secure Encrypted Virtualization (SEV) [75]. SEV was introduced in 2016, and aims to protect VMs from a malicious hypervisor. To achieve this goal, SEV encrypts the VM's memory with a key only known to a special co-processor. This encryption aims to isolate the protected VM's memory by preventing higher privilege adversaries from extracting or injecting plaintext.

However, while SEV does provide memory confidentiality, it refrains from providing memory integrity. This enables high privileged adversaries to modify the encrypted memory. At the same time, it is difficult for the VM to detect such modifications. One may think that modifications of encrypted memory can have little impact on the VM's security. Due to the encryption, the adversary should not be able to change data in a meaningful way, being left only with the ability to corrupt the memory. However, the strong attacker model of a malicious hypervisor gives the adversary numerous possibilities to analyze the VM's behavior. For example, the adversary can track the VM's memory accesses, control its address translation, observe incoming and outgoing network traffic, or interrupt the VM's execution. Despite this great power, the exact abilities of a malicious hypervisor able to modify encrypted memory have not yet been investigated.

To fill this gap, we start by analyzing the impact of a high privileged adversary on a TEE's memory isolation when the TEE provides memory confidentiality, but no memory integrity. To be precise, we show that the adversary can use the missing memory integrity to modify the memory mapping of a VM protected by SEV. Combined with an analysis of the VM's memory access behavior, such a modification permits to utilize a service within the VM to extract plaintext from the VM's memory. In other words, the missing memory integrity allows an adversary in control of the hypervisor to circumvent the VM's memory isolation by violating its memory confidentiality. We further demonstrate that modifications of the VM's memory mapping can even enable the adversary to inject and execute arbitrary code in the VM. For both attacks, we determine the attack vectors required for their execution.

Our attacks reveal that a high privileged adversary able to modify the TEE's memory, even if encrypted, can circumvent the memory isolation and affect both the confidentiality and integrity of the TEE's memory. Hence, memory confidentiality by itself is not sufficient to cryptographically isolate a TEE's memory from high privileged adversaries. Instead, TEEs are required to fend off various attack vectors in order to fully isolate their memory. Only then, they can ensure the confidentiality and integrity of their data and processes. Therefore, we discuss different countermeasures that can be taken by either the VM or SEV and make it possible to prevent the different attack vectors required for successful exploitation. Our most effective countermeasure is a small modification to how SEV encrypts the VM's memory which prevents meaningful remapping of the VM's memory. However, a common property of the most efficient solutions is that they

have to be implemented by the vendor. Hence, we are not able to evaluate the defenses in practice, but focus on their theoretical aspects.

Knowing how to defend against the described attacks enables TEEs to establish a full cryptographic memory isolation, warding off high privileged adversaries manipulating their memory at run-time. This allows us to focus on the second protection mechanism deployed by all TEEs offered by the industry: The attestation of the TEE's initial state [129]. The attestation permits the owner or user of the TEE to verify that it has been launched correctly and that the adversary did not tamper with the provided image [34, 3]. However, the built-in attestation is not able to detect adversaries exploiting vulnerabilities in the TEE's code at run-time. Such vulnerabilities can enable a remote, low privileged adversary to gain complete control over a TEE. To detect the exploitation of such software vulnerabilities, previous work proposes to make use of Control-Flow Attestation (CFA) [1]. CFA analyzes the control flow of a monitored application, the *target*, to detect anomalies indicating attacks. For this, all valid control flows of the target are first collected in an offline phase. Afterwards, in the online phase, the target's control flows are recorded and compared to the flows collected in the offline phase. This allows CFA to detect adversaries modifying the target's original control flow.

Previous work on CFA already makes use of TEE to protect the CFA mechanisms. Yet, only recently the scientific community also started to investigate how to perform CFA when the target itself is running within a TEE [142]. It is exactly this challenge which we discuss in the third part of our work. To be precise, we present a design which permits to perform CFA in environments in which high privilege layers are not trusted. To protect the target from high privileged adversaries, we launch it within a TEE. Additionally, we use a second TEE to verify the control flows of the target. This enables us to protect from high privileged adversaries, while at the same time also being able to detect remote adversaries exploiting software vulnerabilities.

1.1 Research questions

TEEs aim to protect the confidentiality and integrity of code and data within the environment from high privileged adversaries. To achieve this goal, two protection mechanisms are implemented by all TEEs provided by the industry: memory isolation and attestation of the TEE's initial state [129]. The first mechanism can be implemented using cryptographic isolation, which encrypts the TEE's memory with a key inaccessible to the adversary. However, some TEEs attempt to achieve the cryptographic memory isolation by only providing memory confidentiality, but refrain from verifying the integrity of the encrypted memory. Such missing integrity protection can permit an adversary to break the TEE's memory isolation, thereby circumventing this essential protection mechanism.

We demonstrate the practical feasibility of such breaches by presenting attacks from a high privileged adversary against TEEs that do not include integrity protection in their cryptographic memory isolation. Being aware of the impact of missing integrity

protection, we continue by analyzing different approaches on how to prevent the described attacks. The discussed approaches enable the TEE to reliably isolate its memory from high privileged adversaries, thereby ensuring the effectiveness of this essential protection mechanism.

Afterwards, we focus on the second protection mechanism, the attestation. Current attestation mechanisms are not able to protect from security vulnerabilities located in the code running within the TEE. Therefore, we also improve the second protection mechanism by discussing how to extend it in order to detect the exploitation of software vulnerabilities within a TEE. In summary, the analysis and improvement of the two essential protection mechanisms of TEEs, cryptographic memory isolation and the attestation mechanism, leads us to the investigation of the following research questions:

RQ1: Which Attack Vectors Enable Attacks Against a TEE's Cryptographic Memory Isolation Without Memory Integrity? A TEE failing to verify the integrity of its memory can allow a high privileged adversary to violate cryptographic memory isolation and to launch certain attacks even in the presence of memory confidentiality. Hence, we want to explore ways that permit adversaries to attack the TEE despite existing memory encryption. For the discovered attacks, we intend to identify the attack vectors that enable their execution. Furthermore, we want to analyze the impact of the attacks on the confidentiality and integrity of data and processes within the TEE.

RQ2: How Can We Restore Cryptographic Memory Isolation of TEEs Without Memory Integrity? Based on the attack vectors determined in *RQ1*, we intend to analyze countermeasures that prevent them. For this analysis, we want to investigate existing countermeasures from academia and the industry and also propose our own solutions. For each countermeasure, we intend to analyze its requirements, feasibility, performance impact and effectiveness. Since the most efficient approaches would have to be implemented by the vendor, we focus on discussing their theoretical aspects.

RQ3: How Can We Detect the Exploitation of Software Vulnerabilities Within TEEs? With mechanisms to restore cryptographic memory isolation in place, TEEs provide increased protection from high privileged adversaries. However, they will not be able to sustain the confidentiality and integrity of their data and processes in the presence of software vulnerabilities within the TEE's code. Hence, we want to explore how we can extend the built-in attestation mechanisms in order to also detect adversaries exploiting software vulnerabilities within the TEE.

By investigating these questions, we analyze the two pillars on which all industry-provided TEEs base their security guarantees: memory isolation and attestation [129]. With the improved security mechanisms in place, TEEs are able to protect themselves against a wide range of different adversaries, from a malicious hypervisor to a remote user exploiting vulnerabilities within the TEE's code.

Yet, as it is mostly the case in research, the investigation of our research questions will also yield new research questions that should be investigated by future work. First of all, our approach to detect the exploitation of software vulnerabilities within TEEs requires the verifier to be aware of all valid control flows the TEE can execute. Future work is required to investigate how these control flows can be efficiently collected for complex programs. Furthermore, the approach is only able to detect a limited set of non-control data attacks. Such attacks stay within the legitimate control flow, but take unexpected paths (Section 2.3). Also the detection of non-control data attacks that are not covered by our attestation mechanism should be investigated by future work. Finally, we implemented the prototype of our extended attestation mechanism on the TEE Intel Software Guard Extensions (SGX). Future work could investigate how the mechanism could be efficiently implemented on other TEEs, such as AMD SEV Secure Nested Paging (SEV-SNP), under consideration of the TEE's architecture and its specific security features.

1.2 Contributions

To address our research questions, we made different contributions to the field of TEEs. Each contribution addresses one of our three research questions.

1.2.1 Contributions to RQ1

With our first three contributions, we address our first research question. *RQ1: Which attack vectors enable attacks against a TEE's cryptographic memory isolation without memory integrity?*

C1: Attack Vectors for the Extraction of Secrets from TEEs Missing memory integrity in TEEs can allow a high privileged adversary to circumvent memory isolation and to violate the TEE's confidentiality. Using the example of AMD SEV, we show how a malicious hypervisor can exploit missing memory integrity to extract memory content from an SEV-protected VM and analyze the attack vectors that permit the attack. Specifically, we modify the VM's memory mapping and utilize a service running in the VM to return the VM's unencrypted memory pages instead of the original response. To avoid extracting the VM's entire memory, we show how the adversary can focus on memory pages containing critical information such as TLS, SSH or disk encryption keys. This allows us to extract key material from the SEV-protected VM within a short period of time.

Publications: [107, 106]

C2: Attack Vectors for the Execution of Arbitrary Code Within TEEs Missing memory integrity in TEE's can not only affect the TEE's confidentiality, but also its integrity. Sticking to the example of AMD SEV, we show how a malicious hypervisor can execute arbitrary

code within an SEV-protected VM and again determine the underlying attack vectors. For this, we first identify code within the VM, the execution of which we are able to trigger from the outside. Next, we inject a network packet containing arbitrary code into the VM and identify the location of the packet within the VM's memory. Finally, we conclude the attack by again modifying the VM's memory mapping and triggering the execution of the injected code.

Publication: [109]

C3: Framework for TEE Analysis The previous contributions show that missing memory integrity can have a serious impact on a TEE's cryptographic memory isolation by violating the confidentiality and integrity of its memory. To ease testing TEEs for such vulnerabilities, we present a framework for the analysis of encrypted VMs[1]. The framework supports different analysis methods such as memory access tracking and the modification of memory mappings. This will facilitate the security evaluation of upcoming technologies such as Intel Trusted Domain Extensions (TDX) [69], IBM Protected Execution Facility (PEF) [64] and ARM Confidential Computing Architecture (CCA) [8].

Publications: [107, 106, 109]

1.2.2 Contributions to RQ2

Our next contribution investigates our second research question. *RQ2: How can we restore cryptographic memory isolation of TEEs without memory integrity?*

C4: Countermeasure Against the Exploitation of TEEs Without Memory Integrity Our previous contributions show that missing memory integrity can breach a TEE's cryptographic memory isolation. To ward off the demonstrated attacks against SEV-protected VMs, we propose a countermeasure which incorporates the Guest Physical Address (GPA) into the encryption algorithm. This modification will lead to an erroneous decryption when the VM accesses other memory content than intended, for example due to a modified memory mapping. As only the vendor is able to perform such a modification, we analyze the countermeasure on a theoretical level. Additionally, we discuss other countermeasures proposed by academia and the industry.

Publications: [107, 109]

1.2.3 Contributions to RQ3

Finally, with our last two contributions, we address our third research question. *RQ3: How can we detect the exploitation of software vulnerabilities within TEEs?*

[1] `https://github.com/Fraunhofer-AISEC/severed-framework`

C5: Architecture to Retrieve Run-Time Information of Applications from a TEE Applications are often dealing with potentially malicious clients aiming to exploit vulnerabilities in their code. Furthermore, applications hosted in cloud environments are exposed to the threat of a malicious administrator. The latter can be avoided by making use of TEEs. Therefore, we discuss how we can enable a TEE to collect run-time information from an application running outside the TEE based on the example of Intel SGX [34]. This allows us to detect the exploitation of vulnerabilities in the application, while at the same time protecting the monitoring process from high privileged adversaries. We further improve on our concept by presenting an architecture that permits an application to provide run-time information to the monitor while hosting both the application and the monitoring process in a TEE. This architecture allows to retrieve run-time information from the application, while at the same time protecting the application and the monitoring process from a malicious administrator.

Publications: [105, 108]

C6: Porting Control-Flow Attestation to TEEs Having enabled a TEE to collect data of an application running inside another TEE, we show how to use this ability to monitor the application's run-time integrity. For this, we use the already established technique of CFA, and port it to our design which hosts both the application to be attested and the monitor in TEEs. In order to adapt CFA to our use case, we make the mechanism transparent to the potentially malicious end user. Additionally, we apply modifications that allow our CFA mechanism to remain unaffected by Address Space Layout Randomization (ASLR). Finally, we present a technique that permits us to detect Data-Oriented Programming (DOP) attacks by establishing a maximum number of iterations for each loop. Our final design enables us to detect adversaries exploiting software vulnerabilities within TEEs, while at the same time protecting both the application and the monitoring process from a malicious administrator.

Publication: [108]

1.3 Publications

The contributions we describe in this thesis are covered by the following publications. Some publications contain additional contributions not covered in this thesis.

[107] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel. "SEVered: Subverting AMD's Virtual Machine Encryption". In: *European Workshop on Systems Security (EuroSEC)*. 2018

[106] M. Morbitzer, M. Huber, and J. Horsch. "Extracting Secrets from Encrypted Virtual Machines". In: *ACM Conference on Data and Application Security and Privacy (CODASPY)*. 2019

[105] M. Morbitzer. "Scanclave: Verifying Application Runtime Integrity in Untrusted Environments". In: *International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. 2019

[109] M. Morbitzer, S. Proskurin, M. Radev, M. Dorfhuber, and E. Quintanar Salas. "SEVerity: Code Injection Attacks against Encrypted Virtual Machines". In: *USENIX Workshop on Offensive Technologies (WOOT)*. 2021

[108] M. Morbitzer, B. Kopf, and P. Zieris. "GuaranTEE: Introducing Control-Flow Attestation for Trusted Execution Environments". In: *IEEE International Conference on Cloud Computing (CLOUD)*. 2023

Other publications We also made publications which are related to the topic of this thesis, but are not core contributions. These publications focus on the tweak mechanism used for AMD SEV's cryptographic memory isolation and the impact of malicious information provided by a high privileged adversary.

In a first publication, we present the first code execution attack against AMD SEV [148]. Specifically, we reverse-engineer the improved tweaking algorithm applied before the memory is encrypted [75]. With this knowledge, we make use of the missing memory integrity protection and move encrypted blocks within the machine's physical memory. This allows us to create a high-speed encryption oracle, enabling us to execute arbitrary code within VMs protected by SEV, thereby circumventing its cryptographic memory isolation.

In a second publication, we investigate different software vulnerabilities in SEV [124]. Specifically, we show that a malicious hypervisor is able to extract cryptographic information which the VM passes to the `virtio-crypto` engine [124]. We also extract and inject data from and into the VM by setting the `reserved` bit in the VM's page tables. Additionally, by intercepting instructions from the VM to the CPU, we influence the random number generation in the VM, permitting us to practically disable the VM's probabilistic kernel defenses. By using this interception, a malicious hypervisor is able to execute arbitrary code within an SEV-protected VM.

In a third publication, we analyze the impact of the hypervisor's ability to provide the VM with malicious virtual devices [61]. For the analysis, we make use of a combination of static and dynamic methods. By analyzing 22 drivers in the Linux kernel, we identify 48 vulnerabilities which can be exploited by a malicious hypervisor. To underline the impact of such vulnerabilities, we provide a Proof of Concept for two of the discovered bugs, allowing a malicious hypervisor to execute arbitrary code within the protected VM. While we perform the analysis on AMD SEV, we assume that the bugs also apply to similar environments, such as Intel TDX.

[148] L. Wilke, J. Wichelmann, M. Morbitzer, and T. Eisenbarth. "SEVurity: No Security Without Integrity - Breaking Integrity-Free Memory Encryption with Minimal Assumptions". In: *IEEE Symposium on Security and Privacy (S&P)*. 2020

[124] M. Radev and M. Morbitzer. “Exploiting Interfaces of Secure Encrypted Virtual Machines”. In: *Reversing and Offensive-oriented Trends Symposium (ROOTS)*. 2020

[61] F. Hetzelt, M. Radev, R. Buhren, M. Morbitzer, and J.-P. Seifert. “VIA: a Toolset for Analyzing Device Interfaces of Protected Virtual Machines”. In: *Annual Computer Security Applications Conference (ACSAC)*. 2021

1.4 Outline

This thesis is structured as follows. Before describing our contributions in detail, we discuss the required background in Chapter 2. This background consists of an overview of the two TEEs we used for our contributions, namely AMD SEV and Intel SGX. Furthermore, we present the basic concept of CFA, which we deploy in Chapter 6 to detect run-time attacks on TEEs. Afterwards, we discuss the current state of the art in Chapter 3 by examining related work. This permits us to identify the missing gaps which we investigate in this work.

In Chapter 4, we investigate our first research question. Specifically, we present two attacks against AMD SEV’s cryptographic memory isolation in Sections 4.2 and 4.3. These attacks allow a high privileged adversary to exploit SEV’s missing memory integrity in order to violate the confidentiality and integrity of an SEV-protected VM. For both attacks, we determine the attack vectors required for their successful execution.

Having determined these attack vectors, we continue in Chapter 5 by discussing possible countermeasures against the different vectors. By doing so, we investigate our second research question which seeks methods to restore cryptographic memory isolation. Most importantly, we propose a slight modification to SEV that efficiently prevents the attacks described in the previous chapter. Additionally, we provide an overview of the existing countermeasures discussed in different publications and the different attack vectors they address.

Knowing how to restore cryptographic memory isolation, we investigate our third research question in Chapter 6. To be precise, we extend the attestation mechanism of TEEs in Section 6.2 by first presenting an architecture that allows us to collect run-time information of an application running within a TEE. Then, in Section 6.3, we describe a design that combines this architecture with CFA to detect an attacker manipulating the control flow of an application hosted within a TEE. In comparison to the corresponding publication, the design discussed in this work includes a mechanism that enables us to detect DOP attacks by determining the maximum number of iterations for every loop. To underline the practicality of our design, we implemented a prototype, which we describe in Section 6.4 and evaluate in Section 6.5. We then finalize the chapter by discussing details of our prototype in Section 6.6.

Finally, we conclude the thesis in Chapter 7 by summarizing our contributions and discussing possible future work in the field.

CHAPTER 2

Background

Previous work proposed various Trusted Execution Environments (TEEs) [138, 96, 27, 47, 35, 145, 87] which aim to protect code and data from high privileged adversaries. Meanwhile, the industry also offers different TEEs, making them available to a wider audience [17, 34, 75, 120, 69, 64, 8]. In this work, we focus on the two protection mechanisms provided by all of these industry-provided TEEs: memory isolation and attestation [129]. Note that some TEEs may provide additional protection mechanisms for secure I/O or secure storage. Yet, as these are not supported by all TEEs and are not strictly required to achieve the TEE's security guarantees, we focus on memory isolation and attestation.

In this chapter, we discuss the two TEEs which are commercially available at the time of writing and aim to protect lower privileged software from high privileged adversaries: AMD Secure Encrypted Virtualization (SEV) [75, 74] and Intel Software Guard Extensions (SGX) [34]. We use AMD SEV in Chapter 4 to analyze the impact of missing memory integrity on cryptographic memory isolation. In detail, we show that the missing integrity can allow an adversary to violate the confidentiality and integrity of data and processes in TEEs. Afterwards, we use Intel SGX in Chapter 6 to improve the attestation mechanism of TEEs by implementing our design to detect adversaries exploiting software vulnerabilities within a TEE. Additionally to discussing these two TEEs, we also use this background chapter to explain the concept of Control-Flow Attestation (CFA), which plays an important role in the design we describe in Chapter 6. Hence, we describe the basic concept of CFA and its limitations, specifically non-control data attacks.

2.1 AMD SEV

To understand the concept of AMD SEV, we first have to understand the working principles of virtualization. Virtualization allows to run multiple virtual systems, the so-called Virtual Machines (VMs), on the same physical system. The VMs are managed by the hypervisor, a combination of hardware and software mechanisms. One task of the hypervisor is to assist the VM during execution. If the VM requires assistance, it issues the `VMEXIT` instruction, which hands control to the hypervisor. The hypervisor then

saves the VM's state, such as registers, and assists the VM, for example by emulating a CPU instruction. Afterwards, the hypervisor restores the VM's state and hands control back to the VM.

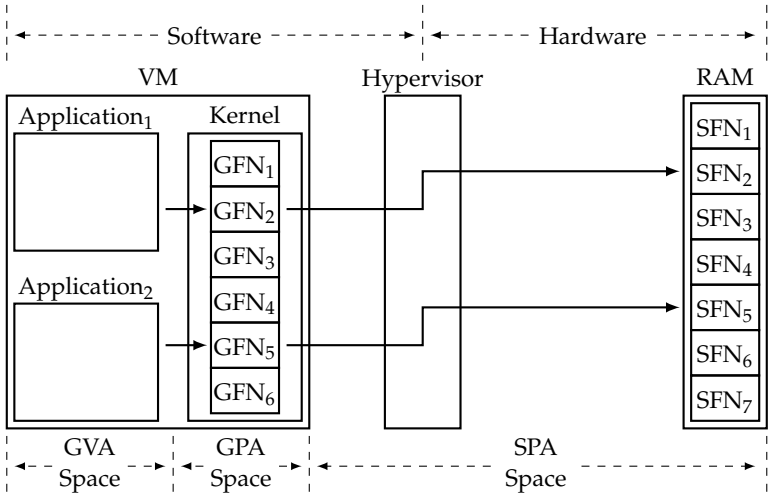

Figure 2.1: Memory management in a virtualized environment. Within the VM, applications access memory via virtual addresses, the GVAs. The mapping between the GVAs and the location in the VM's memory, the GPAs, is managed by the VM. Using SLAT, the hypervisor maps the GPAs to their location in the system's memory, the SPA.

The hypervisor is also responsible for providing virtual hardware to the VM, such as virtual memory. Figure 2.1 depicts how the hypervisor, representing the transition from software to hardware, manages the VM's virtual memory. Depicted on the left, the VM may run different applications, which use virtual addresses to access memory. The VM maps these virtual addresses, the Guest Virtual Adresses (GVAs), to the Guest Physical Adresses (GPAs). These GPAs are used by the VM's kernel to manage the virtual memory provided by the hypervisor, which is treated as physical memory by the VM. This memory is split into different pages, the Guest Frame Numbers (GFNs). In the next step, the hypervisor maps the GFNs to their actual location in the physical memory, the System Frame Numbers (SFNs). To address data within the physical memory, the hypervisor uses System Physical Adresses (SPAs), which point to an offset on a specific SFN. This twofold translation is generally referred to as Second Level Address Translation (SLAT). In Chapter 4, we show how a malicious hypervisor can modify the SLAT to remap a GFN to a different SFN. We use this ability to present our two initial contributions (Section 1.2.1). In the first contribution, we show how the hypervisor can extract secrets from a VM by modifying the SLAT. Furthermore, in our second contribution, we use this ability to execute arbitrary code within the VM.

Another task of the hypervisor is to provide the VM with virtual devices for I/O operations, such as network cards. To exchange data between the physical and the virtual

device, virtualized environments use frameworks such as virtio [44]. In Section 4.3, we track the data exchange process via virtio to determine where the VM stores incoming network packets. For the tracking, it is important to know that the main communication structure of virtio are the so-called *virtqueues*. Each virtual device uses at least two virtqueues: one for receiving data, and one for sending data. The virtio standard defines two different formats for virtqueues: the traditional *split virtqueues*, and the newer *packed virtqueues*. All VMs we analyzed for the evaluation of our code execution attack (Section 4.3.4) made use of split virtqueues, which is why in this work we focus on split virtqueues.

Each split virtqueue comprises three components: the *descriptor table*, the *available ring*, and the *used ring*. The descriptor table contains, among others, the addresses of buffers the virtual device in the hypervisor and the driver in the VM can use for data exchange. When referring to a buffer in the descriptor table, the driver and the device use the position of the buffer in the descriptor table as an index. The driver in the VM writes these indexes into the available ring, thereby offering the buffers to the virtual device in the hypervisor. Once the virtual device has used one or more of the available buffers, it adds the buffer's index to the used ring.

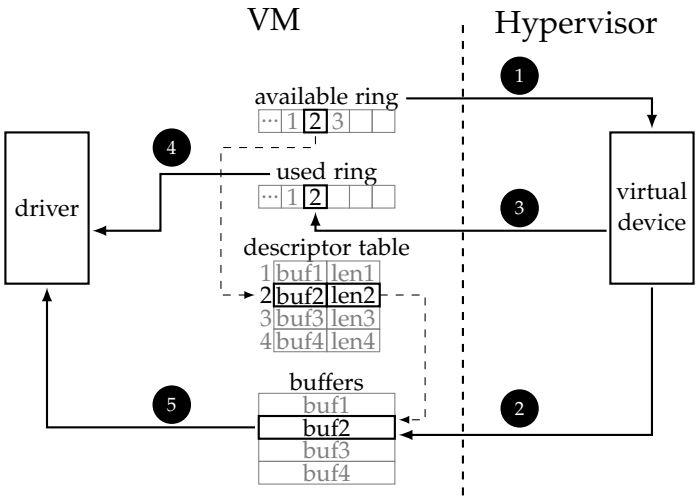

Figure 2.2: Receiving data via split virtqueues. First, the virtual device in the hypervisor reads the available ring and the descriptor table to determine to which buffer to add incoming data. Next, the driver in the VM reads the used ring and the descriptor table to determine the buffer holding the incoming data. Source: [109].

Figure 2.2 depicts handling of incoming data via a split virtqueue. To enable the VM to receive data, the driver allocates buffers, adds them to the descriptor table and writes their respective index into the available ring. When receiving incoming data, the virtual device reads the available ring to find the next free buffer (❶). Afterwards, the device writes the data to the buffer (❷) and adds the buffer's index to the used ring (❸).

Reading the used ring (❹) allows the driver in the VM to determine from which buffer to consume the incoming data (❺).

The process for sending data with split virtqueues works vice-versa. First, the driver puts the outgoing data in one of the buffers indicated in the descriptor table and adds the buffer's index into the available ring. Next, the device reads the available ring to determine from which buffer it can consume the data. Once the device finished consuming the data, it adds the buffer's index to the used ring.

For both sending and receiving data, the hypervisor requires access to the VM's memory to be able to read and write the buffers used to exchange data. This implies that the VM has to trust the hypervisor not to extract or modify data in its memory. To protect from such a threat, AMD introduced Secure Memory Encryption (SME) and SEV.

SME is a hardware extension for both full and partial memory encryption. This hardware extension links the system's page tables with its encryption engine to establish an interface that permits to regulate which memory regions are to be encrypted. Specifically, AMD dedicates a special bit, the `C-bit`, of the system's Page Table Entrys (PTEs) [4] to indicate whether the associated page should be encrypted. The key for this encryption is managed by a dedicated ARM-based processor, the Secure Processor (SP), which generates a new encryption key on each system reset. To encrypt a memory page, the system sets the `C-bit` of the respective PTE, causing the SP to encrypt the respective page via a specially tweaked, high performance AES implementation [75, 148]. The goal of the tweak is to incorporate the SPA of the data into the encryption. This prevents attackers from performing known-plaintext attacks by moving cipher-text blocks in the physical memory. In Chapter 5, we show how the tweak works in detail. Additionally, we present our contribution *C4*. This contribution prevents our attacks against the confidentiality and integrity of SEV-protected VMs (Sections 4.2 and 4.3) by proposing a small modification of the tweak.

A constraint of SME is its limited use in virtualized environments. This is mainly caused by the fact that the memory of VMs and the underlying hypervisor is encrypted with the same key. Thus, the hypervisor can access sensitive data of VMs despite full memory encryption. Addressing this issue is where SEV comes into play.

In Figure 2.3, we give an overview of SEV. Contrary to SME, SEV introduces different isolation domains, and assigns a unique memory encryption key for every domain [75]. In other words, every isolation domain, including different VMs and the hypervisor itself, is associated with a unique secret key. Trying to access encrypted memory from a different isolation domain that is associated with a different key returns garbled information. As such, the physical memory of isolation domains can only be decrypted with the secret key of the particular domain to which the memory has been assigned. Hence, SEV makes use of cryptographic memory isolation to protect its memory. To the best of our knowledge, there are only few examples of TEEs isolating their memory via means of cryptographic instead of logic memory isolation [129], the others being academic works [138, 93]. Being the only industry-provided TEE that uses cryptographic

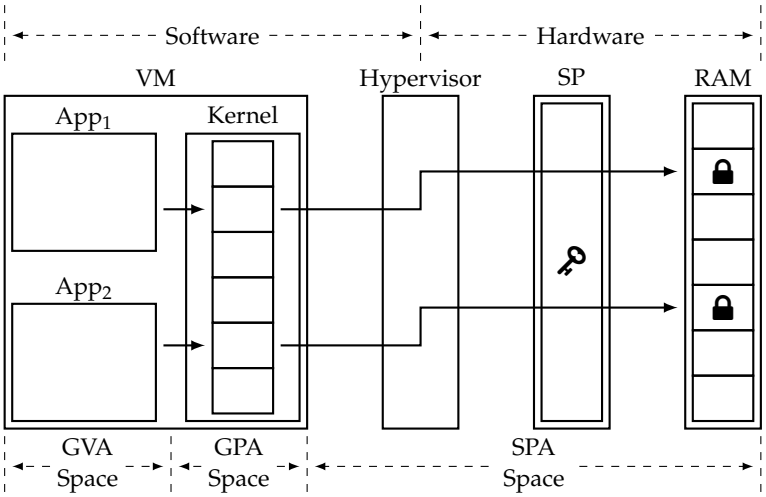

Figure 2.3: An overview of AMD SEV. As with traditional virtualization, the hypervisor maps the GPA to the SPA in RAM. Additionally, the SP encrypts and decrypts data before it is written or read.

memory isolation makes SEV a very well suited object of investigation to study the impact of missing memory integrity on cryptographic memory isolation.

SEV's memory isolation can be activated by setting the `C-bit` in the VM's PTEs. This allows every VM to individually control which pages should be encrypted. If the VM sets the `C-bit` for a page, its content will be encrypted with the VM's key. Such a page is only accessible in plaintext to the VM, and is therefore referred to as a *private page*. By omitting the `C-bit`, the VM defines *shared pages* which are also accessible to the hypervisor. Depending on the `C-bit` in the hypervisor's page tables, shared pages can either be encrypted with the hypervisor's key, or remain unencrypted. In both cases, the VM as well as the hypervisor can access the page.

To enable the hypervisor to exchange I/O data with the VM despite memory encryption, SEV leverages *bounce buffers* [13]. Using bounce buffers, the hypervisor, when providing data to the VM, copies the data into a buffer on a page shared between the VM and the hypervisor. Afterwards, the VM copies the data from the shared page to its destination in the VM's private memory. During this operation, the memory controller automatically encrypts the data when writing it to private memory. In comparison, when the VM wants to provide data to the hypervisor, its device driver first writes the data into a buffer in private memory. From there, the VM bounces it to a shared page, on which it will be stored unencrypted, ready to be consumed by the hypervisor. Using this technique, the VM can communicate with the hypervisor despite the memory encryption, while at the same time removing the requirement for the hypervisor to be able to read and write the VM's memory. In Section 4.3, we track the different steps of this mechanism to determine where the VM stores incoming network packets. This

allows us to send a payload to the VM via a network packet and identify its location within the VM's encrypted memory.

An issue that remains with SEV is that register contents stored on a `VMEXIT` may contain sensitive data, which would leak to the hypervisor [147]. To prevent leakage of register contents, AMD introduced an iterative extension to SEV, SEV Encrypted State (SEV-ES) [74]. SEV-ES additionally lends VMs the capability to encrypt and integrity-protect selected register state every time the VM hands over control to the hypervisor. As the difference between SEV and SEV-ES solely lies in the encryption of the register state, we use the term SEV to refer to both technologies. In comparison, the third iteration of SEV, SEV Secure Nested Paging (SEV-SNP), brings some major changes to the basic design [3]. Therefore, we explicitly refer to the third iteration as SEV-SNP.

The approach taken by the SEV family to protect an entire VM permits to run applications in a TEE without requiring developers to perform modifications. This may very well be the reason why other upcoming TEEs such as Intel Trusted Domain Extensions (TDX) [69], IBM Protected Execution Facility (PEF) [64] and ARM Confidential Computing Architecture (CCA) [8] will also protect entire VMs. Our contribution *C3* provides a framework to analyze these similar technologies.

A downside of protecting a VM is that it creates a large Trusted Computing Base (TCB). The TCB consists of all components the TEE is required to trust. When protecting an entire VM, the TCB includes software such as the kernel and the Operating System (OS), which may consist of hundreds of millions of lines of code [100, 86]. Intel SGX therefore takes a different approach and aims to minimize its TCB by only protecting a single application.

2.2 Intel SGX

Intel SGX was first introduced in 2013 [7, 98, 62]. Its goal is to protect a user-space application from a high privileged adversary. To achieve this goal, SGX splits the application into an untrusted part, the *Host Application (HA)*, and a trusted part, the *enclave*. The HA is a traditional user-space application, and provides no protection against a high privileged adversary. It is responsible for performing non-critical operations and for launching the enclave. During the launch process, the SGX firmware measures all code and data loaded into the enclave. This measurement enables a remote user to verify the integrity of the enclave at launch-time using Intel's remote attestation process [73, 127]. After the enclave is launched, firmware and hardware ensure that its memory cannot be accessed by a higher privilege layer such as the OS by deploying logical memory isolation. Specifically, the CPU prevents non-enclave software from accessing the enclave's memory [34].

The HA and the enclave communicate over a strictly defined interface, the *call gate*. To hand control to the enclave, the HA executes an *ecall*, which calls a function within the enclave. If the enclave during its execution needs assistance from the HA, it issues an *ocall*. Such assistance is for example necessary to perform I/O operations, as the enclave cannot

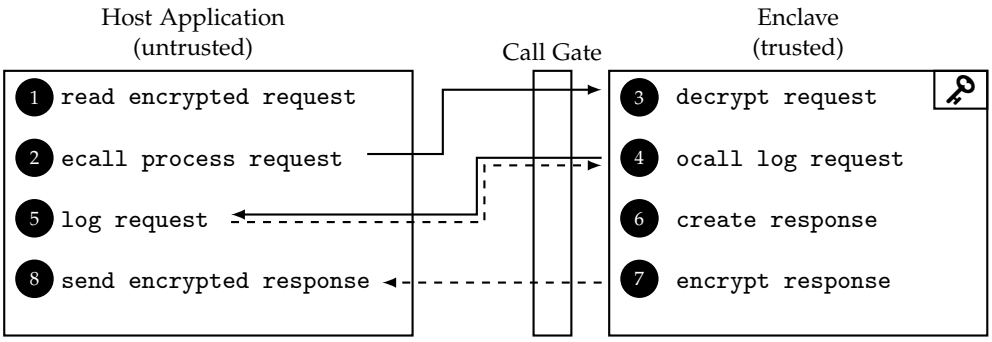

Figure 2.4: An example for the control flow of a server using SGX. The HA receives encrypted requests and hands them to the enclave using an ecall. The enclave decrypts the requests, processes them and performs an ocall for logging.

directly interact with the OS. Both ecalls and ocalls require a context switch, making them expensive operations [146]. Additionally, ocalls flush the Translation Lookaside Buffer (TLB) for security reasons, further increasing the performance costs [98].

Figure 2.4 shows an example control flow of a service using SGX. The two big rectangles depict the HA on the left, and the enclave on the right. Aside from launching the enclave, the HA is responsible for performing non-critical operations, such as handling incoming connections (❶). When the HA receives an encrypted request from a client, it hands the information to the enclave via an ecall (❷). This ecall has to pass the call gate, which ensures the correct invocation of the enclave. Next, the enclave decrypts the request using key material only available within the enclave (❸). Before processing the request, it logs the request (❹). As logging requires writing to a file, the enclave issues an ocall, in which the HA stores the logging information on the file system (❺), and afterwards returns control to the enclave. Note that if the logging information contains critical data, the enclave can encrypt this data before handing it to the HA. Afterwards, the enclave creates a response (❻) and encrypts it (❼). When the enclave is finished handling the response, it returns from the original ecall (❷) to hand control and the encrypted response back to the HA. The HA can then send this response to the client (❽).

While this process is more complex than traditional request handling, the advantage is that the key material stays within the enclave, rendering it inaccessible to high privileged adversaries. This reduces the impact of an attacker gaining control over the environment in which the service is hosted, for example by escaping an isolation domain [103]. Another advantage of this approach is that even when the network handling process contains vulnerabilities, this would only provide access to the HA performing this non-critical operation, but not to the enclave. To gain access to the key material, the adversary is required to also compromise the enclave.

To detect such a compromise, our contribution *C5* describes an architecture to collect run-time information of an application running within a TEE in Section 6.2. We then

utilize this architecture for our contribution *C6*, in which we monitor the run-time integrity of the application by using CFA. This allows us to improve the attestation by extending measurements at time of launch with run-time checks. To underline the practicality of our design, we describe the details of our prototype based on Intel SGX in Section 6.4.

2.3 Control-Flow Attestation

The goal of CFA is to monitor an application's control flow to detect attacks that cause deviation from the normal control flow. This concept was first introduced by Abera et al. in 2016 [1]. Using CFA, the *prover* provides a second party, the *verifier*, with information about the executed control flow of an application, the *target*. This information enables the verifier to determine if the target correctly executed the expected sequence of commands. To ensure the integrity of the verifier even in the case of a compromised prover, the two entities are clearly separated.

```
N1: read input
N2: compare input to password
N3:   if they match, get secret data
N4:   else get public data
N5: return data
```

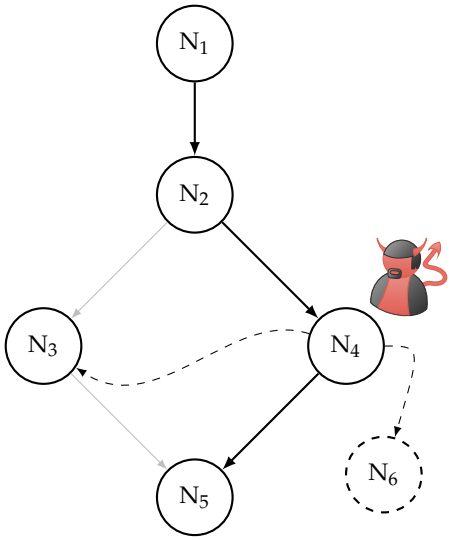

Figure 2.5: An example for legal and illegal control flows in a CFG. Nodes N_1–N_5 denote executions of code, while the arrows denote the different edges. A vulnerability in N_4 can permit an attacker to illegally redirect control flow to N_3 or to a new node N_6. Source: [108].

In order to perform the verification, the verifier first collects all of the target's legal control flows to calculate its legitimate Control-Flow Graph (CFG). In the CFG, the

target's execution is abstractly represented with *nodes* representing uninterruptible instruction sequences and *edges* representing transitions between nodes. Figure 2.5 shows a CFG consisting of the nodes $N_1 - N_5$, and five solid edges marking valid transitions between the nodes. In this example, the target first reads an input in N_1 and compares it to a stored password in N_2. Depending on the result of this comparison, the target will either continue to N_3 if the input and the password match or otherwise to N_4. Finally, both N_3 and N_4 will hand control to N_5, which returns the data fetched in the previous node.

To create the target's CFG, the verifier can make use of static or dynamic analysis. Using static analysis, the verifier analyzes the binary code of the target to calculate the CFG. However, this approach does not include edges that are calculated at run-time, such as indirect calls. In comparison, dynamic analysis attempts to avoid such issues by executing the target, allowing to collect all possible paths of the CFG. For the collection, the verifier executes the target with different inputs and merges all executed paths to create the CFG. Yet, using this approach, it is difficult to determine at which point the target executed all possible control flows at least once. Therefore, efficiently creating complete CFGs for complex programs is still an open research question. To nevertheless ensure that all paths of the target have been invoked, one can resort to programs with reduced complexity.

In the design we describe in Section 6.3, we aim to protect targets with reduced complexity that are frequently deployed in cloud environments, the so-called microservices [6]. Such microservices split the functionality of a complex multi-functional service into multiple, simple services with a single functionality. This reduced functionality also reduces the complexity of each service, simplifying the execution of all legal control flows for each of them. As an alternative to splitting up the target itself, we can break down the CFG of a complex target into multiple segments to facilitate the collection of each segment's CFG, as suggested by previous work [2, 141].

Now, let us assume that N_4 in Figure 2.5 contains a vulnerability which permits an adversary to read and write arbitrary data memory. This would enable the adversary to perform a so-called *control data attack* and, for example, modify a return address on the stack. With the modified return address, the target might then create a new edge by jumping directly from N_4 to N_3. Overwriting the return address would also permit the adversary to create a new node N_6, allowing to execute arbitrary code by performing Return-Oriented Programming (ROP) [132].

CFA can detect such control data attacks. Once the verifier has created the CFG, the prover records the execution path of the target and sends it to the verifier. By analyzing the execution path, the verifier can determine if the recorded path was within the expected CFG. If this is not the case, the verifier can conclude that the target has been compromised. The verifier can then trace back the target's execution path to determine the first mismatching edge, permitting to locate the node in which the compromise has first taken effect.

In our contribution *C6*, we use CFA to detect attackers modifying the control flow of a target running in a TEE. Also placing the verifier in a TEE allows us to protect both the prover and the verifier from high privileged adversary. In Section 6.3, we discuss the challenges when porting CFA to TEE, and our respective solutions.

Yet, traditional CFA fails to detect attacks that stay within the target's legitimate CFG, the so-called *non-control data attacks*. In comparison to control data attacks, non-control data attacks stay within the target's legitimate CFG, but take unexpected paths [119, 28]. An example for such an attack is the modification of a decision-making variable, such as the password in N_2 in Figure 2.5, to circumvent authentication. More advanced non-control data attacks, the so-called Data-Oriented Programming (DOP) attacks, chain arbitrary code gadgets while staying within the CFG [63]. Such DOP attacks require a *gadget dispatcher*. A gadget dispatcher is a loop containing a control structure which permits to select the DOP gadgets to be executed. To be able to chain DOP gadgets, the attacker modifies data such as the loop's induction variable. Modification of this variable allows to perform an arbitrary number of loop iterations, executing only specific DOP gadgets each time. This enables the adversary to perform malicious, Turing-complete computations while staying within the CFG.

Their compliance with the CFG makes it difficult to detect non-control data attacks. Previous work proposes bounds checking to detect variables being overwritten, allowing for the combined detection of non-control data and control data attacks [38]. However, as a considerable disadvantage, bounds checking adds a significant overhead to every read and write operation. Another possibility to detect non-control data attacks consists in storing hashes of critical variables in a trusted area when the variables are written [139]. Afterwards, when the variables are read, a verification mechanism compares the value of the variable with its stored counterpart to detect modifications. However, also this approach requires bounds checking in order to detect writes to critical variables via pointers. In Section 6.3, we show how we can avoid the need for bounds checking by adopting a lightweight approach which enforces an upper limit on loop iterations at run-time. This approach allows us to detect DOP attacks by using CFA.

CHAPTER 3

State of the Art

In this chapter, we discuss related work regarding attacks on Trusted Execution Environments (TEEs) and their defenses as well as work on memory integrity monitoring. Our goal is to determine the current state of the art and to identify the missing gaps which we fill with our work. To achieve this goal, we start with work that describes attacks on AMD Secure Encrypted Virtualization (SEV) and give an overview of all published attacks. We then continue with work that discusses defense mechanisms that could be applied to prevent these attacks. Afterwards, we examine work which aims to determine the memory integrity of various systems. For each section, we also identify the gap covered with the contributions in this thesis.

3.1 Attacks Against AMD SEV

When we started analyzing the impact of missing memory integrity on cryptographic memory isolation, most attacks against TEEs targeted Intel Software Guard Extensions (SGX). However, to the best of our knowledge, all attacks against SGX are side-channel attacks [112], which do not exploit vulnerabilities in the TEE itself. Since AMD explicitly excludes such attacks from SEV's threat model [3], we instead focus on attacks against AMD SEV.

Hetzelt and Buhren [60] were the first to perform an in-depth analysis of SEV. In their work, they use the missing integrity protection and combine it with hypervisor's ability to modify the Second Level Address Translation (SLAT) tables. By analyzing system call sequences, the authors are able to modify the Virtual Machine (VM)'s login information for an SSH server in order to gain unauthenticated access to the VM. Yet, their attack has a success rate of only 23%. This low rate is mainly caused by the fact that for the attack to succeed, the login information of the victim and the attacker have to be stored on the same offset within a Guest Frame Number (GFN). Another disadvantage is that the attack requires an in-depth analysis of the target, for example by having access to a comparable VM. To perform such an analysis, the same authors also discuss using machine learning to understand the behavior of the VM, which also requires a significant amount of preparation [20].

Du et al. [41] analyze the encryption algorithm of SEV. They determine that before encryption, the plaintext is tweaked with static values depending on the System Physical Address (SPA) on which the data will be stored. This knowledge enables them to gain access to an SSH server running within the encrypted VM by analyzing cipher blocks. The tweak values published by the authors are also used by Li et al. [91], who build a decryption oracle. Specifically, they replace encrypted memory blocks about to be sent via an SSH connection with arbitrary blocks from the same VM. This allows them to read the block's content once the packet was copied out of the VM. Similarly, they create an encryption oracle by replacing data in the shared memory before it is bounced to the encrypted memory of the VM. However, a disadvantage of both attacks is that they require the active use of an SSH server within the target VM. Additionally, both attacks only apply to the first generation of AMD EPYC CPUs, as we discovered that the tweaking algorithm was changed in later versions [148].

Werner et al. [147] monitor register values and memory accesses of an encrypted VM to derive and modify instructions the VM executes. Yet, this approach cannot be applied to SEV Encrypted State (SEV-ES), as SEV-ES prohibits the hypervisor from accessing the VM's register values. In the same work, the authors also use Instruction Based Sampling to gather information about which applications are running within the encrypted VM. While this analysis can also be applied to VMs protected with SEV-ES, it has only limited impact on the VM's confidentiality, and no impact on its integrity.

Li et al. [90] modify the Address Space Identifier of a VM during a `VMEXIT`, permitting them to decrypt the page tables of the target VM. By additionally modifying the VM's state, the authors are able to create decryption and encryption oracles. As SEV-ES prevents such a modification of a VM's state, their attack cannot be applied to VMs protected by SEV-ES.

Wilke et al. [149] analyze SEV's attestation process and discover that the order of blocks in the initial image does not influence the attestation measurement. Using this knowledge, they are able to execute arbitrary code within the VM by modifying its image in order to redirect the execution of code. To permit the detection of such modifications, AMD updated the attestation process to also include the VM's memory layout. This enables the owner of the VM to detect a modified VM image during the attestation process.

Li et al. [89] monitor the ciphertext of register values the VM stores on a `VMEXIT` before handing control to the hypervisor. This permits them to infer the VM's behavior and to extract the private keys used by OpenSSL. Similar to Hetzelt et al. [60], the attack requires access to a comparable VM to acquire the detailed knowledge of the VM and the target application necessary to perform the behavioral analysis.

Additionally, there exist attacks against the Secure Processor (SP). One of these attacks uses a bug in the SP to extract the processor's private key [26]. Similarly, also glitching can be used to extract vital keys from the SP [21]. Other hardware attacks replace the firmware, allowing to circumvent the attestation process [116, 22].

Table 3.1: Overview of all software attacks against SEV and SEV-ES in chronological order. The table lists the type of attack and its main requirement. The publications discussed in this chapter as well as in our additional publications are highlighted in bold.

Work	Type of Attack	Requirements
Hetzelt and Buhren [60]	Control-Flow Modification	User Interaction
Du et al. [41]	Control-Flow Modification	Specific CPU Generation
Morbitzer et al. [107, 106]	**Data Extraction**	**In-VM Service**
Werner et al. [147]	Data Extraction	No SEV-ES
Li et al. [91]	Data Extraction and Injection	Specific CPU Generation
Wilke et al. [148]	**Code Execution**	**Specific CPU Generation**
Li et al. [90]	Data Extraction	No SEV-ES
Radev and Morbitzer [124]	**Code Execution**	**Software Bugs**
Morbitzer et. al [109]	**Code Execution**	**I/O Channel**
Wilke et al. [149]	Code Execution	Boot Time Access
Li et al. [89]	Data Extraction	Behavioral Analysis
Buhren et al. [21]	Firmware Key Extraction	Physical Access
Hetzelt et al. [61]	**Code Execution**	**Software Bugs**

Table 3.1 gives an overview of all software attacks against SEV. In bold, we highlight the attacks discussed in the following chapter [107, 106, 109] as well as attacks covered in our additional publications [148, 124, 61]. The overview shows that we were the first to present a generalized approach for attacks against missing memory integrity by providing a data extraction attack against SEV-protected VM's. Additionally, this was the first attack evaluated on a working SEV-setup. Furthermore, we were able to provide a total of four code execution attacks [148, 124, 109, 61].

In this work, we systematically analyze the attack vectors that allowed us to execute our data extraction and code execute attack. Having gained knowledge about the attack vector will permit us to determine the impact of different countermeasures in Chapter 5.

3.2 Defenses for AMD SEV

Along with the different attacks on SEV, also a number of defense mechanisms have been proposed and were partially also implemented. From the attacks described in the previous sections, two code execution attacks were addressed with firmware and software updates [148, 124].

A countermeasure from Li et al. [91] targets the fact that with SEV, the VM copies I/O data such as network packets from the shared, unencrypted into private, encrypted memory before processing it (Section 2.1). This permits a malicious hypervisor to observe the copy process and to determine to which destination the VM copies the data. To

eliminate this weakness, the authors suggest to enable the IOMMU to write the data directly into the VM's memory using the respective encryption key. However, in this scenario, the SEV firmware must ensure that higher privileged software does not abuse this functionality, which will be difficult.

Another attack vector on which multiple attacks rely is the fact that the hypervisor is able to remap GFNs of an SEV-protected VM to a different System Frame Number (SFN). To prevent such remapping, the untrusted hypervisor can be liberated from the task of managing the VM's memory. Based on this idea, Zhang et. al [154] and Li et al. [92] propose to split the hypervisor in a trusted and an untrusted part. In their designs, most tasks are performed by the untrusted part, while the trusted part is, among others, responsible for memory management. An attacker able to gain control over the hypervisor will likely only gain control over the untrusted part, as this part performs a higher number of tasks therefore and has a bigger code base. However, control over the untrusted part of the hypervisor will not allow to remap a GFN to a different SFN, as only the trusted part of the hypervisor can perform such modifications. Similarly, Wu et al. [150] aim to prevent remapping attacks in SEV by introducing an additional component responsible for managing the VM's memory. However, both approaches only outsource trust to other components rather than addressing the attack vector caused by the missing integrity protection.

In comparison, adding integrity protection for the VM's memory would eliminate the pillars of both our attacks as well as others [60, 41, 148, 91, 90]. The best solution for such a protection would be a full-featured integrity and freshness protection of the VM's pages additional to the encryption, as realized in Intel SGX [34]. Therefore, Li et al. [91] propose to ensure the integrity of the VM's memory by storing a hash value for every encrypted block. This approach would permit to detect if data in memory has been modified before it is used by the VM. However, such integrity protection requires high amounts of storage to retain all hash values, even though the authors propose to decrease the storage overhead by using techniques such as Address Independent Seed Encryption in combination with Bonsai Merkle Trees. Alongside the storage overhead, such a solution would likely also induce a high performance overhead, as the integrity of the data would have to be verified on each memory access.

Instead, the latest iteration of SEV, SEV Secure Nested Paging (SEV-SNP), additionally stores the Guest Physical Address (GPA) to protect the integrity of the VM's memory [3]. Specifically, SEV-SNP adds a *Reverse Map Table (RMP)* to the address translation process. As with SEV and SEV-ES, the VM translates a virtual address to a GPA, and afterwards the hypervisor uses the SLAT to translate the GPA to the SPA. Yet, in contrast to previous versions, with SEV-SNP, the SP afterwards verifies the integrity of the translation with the help of the RMP. To perform this verification, the RMP is indexed using the SPA. Each entry in the RMP contains among others information to which VM the SPA belongs to, the matching GPA, and a *valid* flag. If an attacker modifies the SLAT to map a GPA to a different SPA, the SP will be able to detect this modification and the address translation will fault with a `#PF` exception. This allows SEV-SNP to prevent us from remapping the

GFNs of protected VMs, thereby stopping both attacks outlined in the following chapter. Yet, note that this latest version of the SEV family was not yet available when our attacks were originally developed.

In comparison to these works, we systematically analyze how the various attack vectors exploited by the attacks can be prevented. For this, we discuss countermeasures for the different attack vectors which can be taken by both the VM and SEV. Most importantly, with our contribution *C4*, we propose a lightweight countermeasure that prevents a malicious hypervisor from mapping GFNs of a SEV-protected VM to different SFNs in a targeted manner. Additionally, we compare the countermeasures discussed by related work with the ones proposed in this thesis.

3.3 Memory Integrity Monitoring

Also a number of integrity monitoring mechanisms have been proposed in the past. Yet, the only other work we are aware of that performs run-time attestation of TEEs is by Toffalini et al. [142]. To protect their verifier against attacks from a compromised prover, the authors move it to a separate host. For the attestation, they assume the target to be simple enough to be modeled with symbolic execution and insensitive static analysis they use to extract actions from the target. This static approach cannot include indirect jumps calculated at run-time.

Other related work either refrains from using TEEs, or only uses them to protect the monitoring process. In detail, Kil et al. [77] were the first to introduce the notion of dynamic attestation. To perform their attestation, the authors attest two dynamic system properties proposed in earlier work. The first property is structural integrity, which makes use of the fact that dynamic memory objects follow certain patterns [78]. Furthermore, they attest the integrity of global variables by ensuring that they satisfy data invariants [50]. For their work, the authors assume a trusted Operating System (OS). Additionally, their prototype only performs its attestation measurements when the monitored application invokes a system call, giving the attacker the possibility to bypass the detection.

Also Davi et al. [37] propose an architecture for dynamic integrity measurements. To monitor the integrity of applications, the authors suggest to use taint analysis and dynamic tracing. Yet, they leave the detailed implementation of those approaches up to future work. Furthermore, the authors also require a trusted OS.

Rein [125] monitors the integrity of certain segments of ELF binaries in memory. This allows to detect attacks such as code injections, code manipulations, and code pointer modifications in the data segment jump tables. The monitoring is limited to segments with predictable content, such as the `.text` or `.got` segment. Due to this limitation, it is only possible to detect attacks that modify the application in memory but not code reuse attacks such as Return-Oriented Programming (ROP).

To detect attacks such as ROP, Abera et al. [1] introduced the concept of Control-Flow Attestation (CFA). For each legal control flow within the target program, they create a

hash chain which contains the addresses of every executed basic block. While the hash chain enables the prover to store an entire control flow in a single hash, it also requires the verifier to store a hash for every possible flow. Furthermore, they use addresses to identify basic blocks, which prevents the target from using Address Space Layout Randomization (ASLR). To ensure the integrity of the hash chain, they implement their Proof of Concept on a Raspberry Pi 2 using ARM TrustZone. Similar to other works, also these authors do not consider an attacker to be in control of the OS running the target program. Furthermore, their design cannot be applied to systems making use of ASLR.

Dessouky et al. [39] build on the work of Abera et al. [1], but implement their design in hardware. This permits them to reduce the overall overhead and to attest the control flow without having to instrument the target program. Another hardware-based design by Zeitouni et al. [153] additionally provides protection from physical attacks. Both Dessouky et al. [39] and Zeitouni et al. [153] realize the respective advantages by monitoring all executed instructions. Further work by Dessouky et al. [38] attempts to reduce the complexity when creating and verifying the Control-Flow Graph (CFG) by splitting it up into segments. Additionally, their concept allows to detect Data-Oriented Programming (DOP) attacks by analyzing all accesses to data memory. Yet, this analysis of all data memory access also causes additional overhead. Furthermore, all of these concepts require changes to the hardware, which are difficult to apply in cloud environments.

Abera et al. [2] use CFA to detect misbehavior of devices in distributed systems. In their work, they only attest critical code, reducing the performance overhead required to perform CFA. Additionally, they reduce the storage overhead by using multiset hash functions. Also Conti et al. [33] apply CFA to distributed systems, enabling them to detect services that perform non-intended operations. They achieve this goal by presenting a protocol which allows for the attestation of a control flow over multiple devices. Another design for distributed systems by Koutroumpouchos et al. [83] proposes the application of Berkeley Packet Filters to extract the CFG from devices.

Sun et al. [139] use CFA to verify the integrity of an operation executed on an embedded system. To reduce the overhead of the CFA, they split the attestation report into a trace giving information about forward edges and a hash chain of return values ensuring backward edge protection. To prevent DOP attacks, the authors store hash values of critical variables in a trusted area each time the variables are written. On every read of such a variable, they verify the variable's value by comparing it to the previously stored hash. To ensure the inclusion of pointers to critical values, they additionally apply a memory safety approach which drastically increases the overhead. A limitation of this design is that its backward edge protection is not compatible with ASLR. Additionally, their mechanism requires memory safety techniques to detect DOP attacks, causing additional overhead. Furthermore, again the authors only use TEEs to protect the recorded measurements, not to protect the target itself.

Control-Flow Integrity (CFI) mechanisms take a slightly different approach and verify edges before their execution [24, 23, 157]. This requires to store meta data about the legal

edges, which needs to be protected from an attacker. Existing CFI mechanisms therefore aim to hide this information. However, previous work showed that CFI can still be circumvented by overwriting the meta data hidden within the same isolation domain [52, 54, 115]. Considering the much stronger threat model of a malicious administrator, securely hiding the meta data becomes even more difficult.

Toffalini et al. [141] move the meta data required for the control-flow verification into the verifier by introducing remote shadow stacks. Furthermore, they reduce the complexity of the CFG by splitting it into segments and reducing it to only contain nodes critical for the target's execution. Similarly, also Zhang et al. [156] move the control-flow verification into the remote verifier. Additionally, they simplify recorded control flows of complex systems by skipping certain direct calls and folding control-flow events such as loops and recursions. As done by previous works, also Toffalini et al. [141] and Zhang et al. [156] use the kernel as trust anchor, making their design vulnerable to a high privileged adversary. However, both works also mention CFA under a compromised operating system as future work, thereby identifying the need for such a solution.

Ge et al. [53] and Azab et al. [9] use ARM TrustZone to monitor the integrity of a kernel running in the untrusted world from the TEE. TrustZone facilitates such a monitoring process, as it gives the TEE full control over the untrusted part of the system. Similarly, Zhang et al. [155] and Petroni et al. [118] use a secure coprocessor for integrity verification. The mechanisms of these works were designed for administrators wanting to verify the integrity of their systems. Yet, it is difficult to apply them to scenarios in which not only a malicious client, but also a malicious administrator needs to be considered.

What these works have in common is that they are not designed to protect against a malicious administrator. It is exactly this gap which we aim to close by hosting not only the monitoring process, but also the process to be monitored in a TEE. Additionally, we present a previously unpublished, lightweight detection mechanism for DOP attacks incorporated into the CFA. We achieve this goal by detecting whether the number of loop iterations exceeds a previously inferred maximum number of iterations.

3.4 Summary

In this chapter, we discussed various attacks on AMD SEV exploiting different attack vectors. While countermeasures were proposed that may prevent some of these attack vectors, related work does not provide a systematical analysis of which attack vectors could be prevented with which countermeasures.

Furthermore, previous work also discussed solutions to monitor the memory integrity of different targets. However, they rarely consider the threat model of a malicious administrator, which is of increasing importance especially in cloud environments.

CHAPTER 4

Analyzing the Impact of Missing Memory Integrity by Example

By aiming to prevent attacks from high privileged adversaries, Trusted Execution Environments (TEEs) are exposed to a very strong attacker model. To comprehend the power of such an attacker, we investigate our first research question in this chapter, *RQ1: Which attack vectors enable attacks against a TEE's cryptographic memory isolation without memory integrity?*. To identify these attack vectors, we systematically analyze the possibilities of a high privileged adversary targeting a TEE protecting a Virtual Machine (VM). As concrete example, we use AMD Secure Encrypted Virtualization (SEV) (Section 2.1), which was the first publicly available TEE of this type. Yet, meanwhile also other TEEs protecting VMs have emerged, such as Intel Trusted Domain Extensions (TDX) [69], IBM Protected Execution Facility (PEF) [64] and ARM Confidential Computing Architecture (CCA) [8]. These TEEs are faced with similar attack vectors, making our analysis also relevant for these latest developments.

SEV deploys cryptographic memory isolation by giving VMs the ability to encrypt their memory using a key maintained by a dedicated Secure Processor (SP). Consequently, without knowing the respective secret key, neither an attacker performing cold boot attacks nor the hypervisor should be able to reveal the VM's memory contents. However, while SEV encrypts the VM's memory, it does not verify the integrity of the encrypted memory.

We show that this design choice can have a severe impact on the TEE's memory isolation, specifically on the confidentiality and integrity of its memory. In particular, we present our first two contributions, *C1* and *C2*. While *C1* attacks the confidentiality of SEV-protected VMs, *C2* attacks their integrity. Along with the description of the attacks, we collect the different attack vectors that allow us the execute the attacks. We will use them in Chapter 5 to discuss different countermeasures aiming to prevent the different attack vectors.

For both contributions described in this chapter, we combine the elaborated analysis possibilities of a malicious hypervisor with the exploitation of SEV's missing memory integrity. In detail, for one and the other we first identify critical data in the VM's

memory by tracking its accesses to memory or disk and observing its network traffic. These analysis methods represent our first set of attack vectors. Afterwards, we exploit the fact that while the VM controls the translation from Guest Virtual Adresses (GVAs) to Guest Physical Adresses (GPAs), the hypervisor remains in charge of mapping the GPAs to their System Physical Adresses (SPAs) in main memory. In combination with SEV's missing integrity protection, this enables us to change the VM's memory layout from the hypervisor. This change in memory layout is our most important attack vector. For *C1*, we capitalize on this vector and utilize a service in the VM, such as a web server, to provide us with pages containing valuable secrets of the VM in plaintext. Similarly, for *C2*, we let the VM execute a payload we inject via a network packet. We achieve this by mapping code which we can trigger to be executed from the outside of the VM to the injected payload. For this attack, we identify two more attack vectors, namely the hypervisor's ability to inject interrupts into the VM and SEV's unprotected DMA channel. To determine the severity of the attacks, we provide a detailed evaluation of the described attacks in different scenarios, all showing a very high success rate.

It is important to note that our attacks vectors are no software bugs, but fundamental vulnerabilities in SEV's design. In other words, the pillars upon which we base our attacks are completely independent from implementation details such as the Operating System (OS) or the I/O virtualization framework. Thus, targeting another OS or framework would not change the basic steps from any of our attacks.

Along with *C1* and *C2*, we also introduce our third contribution, a framework for the analysis of encrypted VMs[1]. The framework allows to perform the different steps of both attacks we describe in this chapter. This will facilitate the analysis of similar upcoming TEE technologies such as Intel TDX [69], IBM PEF [64] and ARM CCA [8].

4.1 Attacker Model

For the attacks described in this chapter, we assume the target TEE to be AMD SEV. Within the SEV environment, the adversary is in control of the hypervisor. Whether the hypervisor has been compromised or intentionally set up for malicious purposes is out of scope. Our goal is to breach the confidentiality and integrity of a VM protected by SEV's cryptographic memory isolation. This isolation prevents the adversary from directly reading or writing meaningful data into the VM's memory.

To be able to breach the VM's confidentiality (Section 4.2), we assume that the VM offers a service via a publicly accessible remote connection. An example for such a service could be an HTTP, SSH, FTP or mail server. The service provides a resource from the VM's memory, such as an HTML page or a file offered for download. This resource can be spread over one or more pages in memory. We use the resource to extract secrets stored in the VM's memory which we assume are not protected by additional mechanisms such as obfuscation. As concrete examples, we target private TLS keys from

[1] `https://github.com/Fraunhofer-AISEC/severed-framework`

the Apache and nginx web server, private SSH keys, and secret Full Disk Encryption (FDE) keys.

To breach the TEE's integrity (Section 4.3), we do not require any service inside the VM, network communication or disk I/O. Instead, we only assume that the attacker has access to at least parts of the VM's kernel binary. For the kernel itself, we expect a Linux kernel, the specific version of which is irrelevant. This is due to the fact that we do not rely on any vulnerabilities in the kernel, but instead exploit SEV's missing memory integrity. Hence, we expect our attack to also work with future versions of the Linux kernel. For the kernel, we assume that Kernel Address Space Layout Randomization (KASLR) is enabled. However, our attack will also work without this mechanism. In fact, missing KASLR would make the attack easier, as we would not have to bypass it (Section 4.3.1). Aside from running a Linux kernel, we expect the VM to deploy virtio for communicating with the system's devices, the specific version of which again is not important. While we focus on exploiting the network communication channel, we do not require any service in the VM to attack its integrity. Additionally, we disregard any in-VM IP or port filtering mechanisms, and are also agnostic to any in-VM services or open ports. To be precise, we are also not limited to exploiting the network communication channel, but could for example also target disk I/O. Finally, we rely on AMD's event injection interface to execute virtual interrupts at known registered locations.

4.2 Impact of Missing Memory Integrity on the Confidentiality of TEEs

In this section, we want to showcase the extensive control and analysis possibilities of a malicious hypervisor over a VM, even if it is encrypted. We will see that these possibilities in combination with security weaknesses, such as AMD SEV's missing integrity protection, can allow a malicious administrator to perform attacks against encrypted VMs. Previous work has already shown that SEV's missing integrity protection could enable a high privileged adversary to interfere with the protected VM's control flow [60, 41]. In addition to these works, we now present our contribution *C1*, an attack that breaches the confidentiality of a VM protected by SEV. Our goals are twofold: For one, we want to highlight the challenges when aiming to protect against an untrusted hypervisor and showcase the impact of an attacker that can combine the extensive analysis possibilities with security weaknesses. Second, we want to identify the required attack vectors in order to develop and discuss countermeasures that can prevent them. To perform our memory attack against the VM's confidentiality, we make use of a service running in the VM, which uses data stored in the VM's memory to create its responses. Specifically, we perform the following three steps depicted in Figure 4.1:

1. **Resource Identification.** Identify data in the VM's memory storing information served by a service, our *resource*.
2. **Secret Identification.** Determine the page holding the *secret* to be extracted, such as a private TLS or SSH key or a secret FDE key.

❸ **Secret Extraction.** Modify the Second Level Address Translation (SLAT) to point the page holding the resource to the page holding the secret, and extract its content.

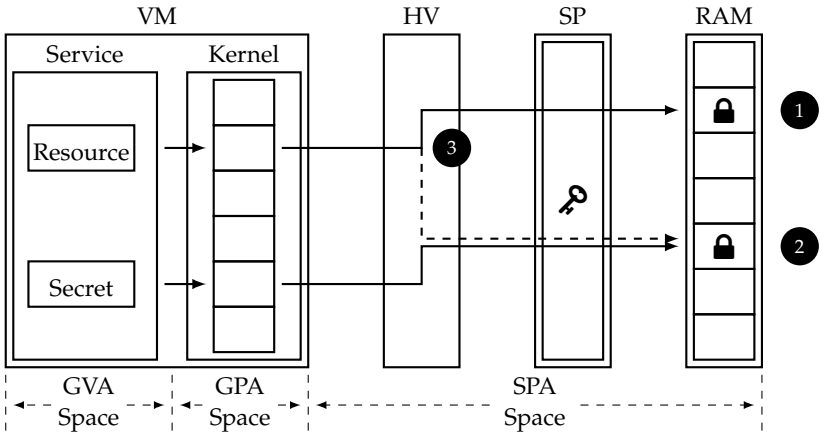

Figure 4.1: Attacking the VM's confidentiality. After identifying a resource and a secret within the VM's memory, we modify the SLAT to extract the secret.

In the first step, we use attack vector *AV1: Page Access Tracking* to track the VM's page accesses in order to identify pages accessed by the service when processing a request. By then combining the access patterns of multiple requests, we are able to identify the resource. This resource is stored in the VM's memory and, most importantly, contained in the response issued by the service. Furthermore, we combine this attack vector with *AV2: Network Monitoring* and *AV3: Disk Access Monitoring* to identify memory pages holding valuable secrets, such as disk encryption keys or TLS keys. To be precise, we track the VM's page accesses while triggering it to access the secret, allowing us to identify a limited set of candidate pages holding the secret. Finally, we abuse SEV's missing integrity protection in *AV4: SLAT Remapping* and modify the SLAT managing the mapping between Guest Frame Numbers (GFNs) and System Frame Numbers (SFNs) (❸). Specifically, we map the GFN containing the resource to the SFN containing the secret to be extracted. This will cause the service to respond with the secret when we request it to provide the resource.

4.2.1 VM Page Access Tracking

An important attack vector for both our attacks is *AV1: Page Access Tracking*. For AV1, we track the pages accessed by the VM, permitting us to identify both the resource and the secret. To perform this tracking, we modify the Page Table Entry (PTE) in the SLAT of the respective pages. Specifically, each PTE provides multiple descriptors used for definition, protection and segmentation [4]. One of those descriptors is the `Present (P)` bit, which indicates if the content of the respective page currently resides in memory, or is swapped out. In case the VM attempts to access a page swapped out of the physical memory, the hypervisor gets notified to reload the page. We profit from this behavior in

our *access tracking* mechanism by clearing the P bit for all of the VM's pages we want to track. This will cause every access to a tracked page to trap in the hypervisor, allowing us to determine which page the VM is about to access. Additionally, we can retrieve the information whether the VM attempted a read, write or execute access. Having determined an access to a tracked page, we restore the respective P bit, permitting the VM to continue its normal operation. Therefore, our access tracking records each accessed page exactly once.

Similarly, we use the `Read/Write (R/W)` bit indicating if a page is writable for our *write tracking* mechanism. An attempt of the VM to write to a page with a cleared R/W bit will trap in the hypervisor, which can then perform the appropriate actions. Therefore, clearing this bit for one of the VM's pages lets us get notified as soon as the VM attempts to write to this page. In other words, in comparison to the access tracking, the write tracking will only cause a trap for a specific type of access, a write access. Having recorded the trap, we restore the R/W bit in order to allow the VM to continue its operation.

Our last tracking mechanism is *execute tracking*. For this mechanism, we make use of the `No Execute (NX)` bit marking a page as non-executable. Similar to the write tracking mechanism, the execute tracking notifies us for one specific type of access. To be precise, the execute tracking will trap when the VM attempts to fetch instructions from a tracked page.

In summary, our three different tracking mechanisms allow us to perform access, write or execute tracking on a given set of the VM's pages. We included all of them in our framework for TEE analysis[2], our contribution *C3*. The tracking mechanisms work on page granularity, which typically corresponds to four kilobyte [4]. To further increase precision, we could extract the VM's register contents at each trap to determine at which offset the VM accesses the page. However, we cannot apply this method to SEV Encrypted State (SEV-ES), as it encrypts the VM's registers before handing control to the hypervisor [74]. Therefore, we refrain from accessing the VM's registers in order to provide a general approach that applies to both SEV and SEV-ES.

4.2.2 Resource Identification

Using *AV1: Page Access Tracking*, we start by identifying the resource in the VM's memory used by the service to send responses (❶). To facilitate the subsequent extraction process, we aim to use a resource fulfilling certain requirements. These requirements include the size of the resource, concurrent accesses by other parties, and its probability to remain at the same GFN.

The size of the resource determines whether we can extract all content of a page, and how many requests we require for extraction. Specifically, to be able to extract data from all possible page offsets, the resource must cover at least one entire page. Yet, the resource does not have to be page-aligned and can, for example, cover the second half

[2] `https://github.com/Fraunhofer-AISEC/severed-framework`

of one page and the first half of another page. Nevertheless, it is still preferable to avoid such resources, as the parts of the pages not covered by the resource might be accessed concurrently in the VM. Such concurrent accesses may lead to unexpected behavior during the extraction process. To exclude such behavior, we prefer resources which cover one or more complete pages.

Another important factor about the resource is its probability to remain at the same GFN during the whole attack. This probability can be influenced by several factors, such as the type of resource, the priority of the respective process, and the VM's memory pressure. One type of resource are so-called file-backed resources, which cache part of a file in memory. This attribute plays an important role in situations in which the VM is low on memory and therefore needs to identify pages it can evict from memory. By default, Linux systems prioritize file-backed pages over non-file backed pages when choosing pages for eviction. This behavior increases the probability for file-backed pages to remain at the same GFN.

During our tests, we found that resources representing the content of a file in memory are especially convenient. For one, they have the advantage that a sufficiently large file located in memory always covers at least one memory page. Additionally, they typically do not share the pages with other data. Therefore, such resources always start at page offset zero, facilitating extraction.

Having decided on a resource, we identify the resource in the VM's memory. For this identification, we have to consider that the resource can be located on a single page, or on multiple pages. Now, let P be the set of all of the VM's pages. Then, we define the unknown set of resource pages as

$$R \coloneqq \{p \in P : \text{Page } p \text{ contains (part of) the target resource}\}$$

To determine R, we activate access tracking on all of the VM's pages. Next, we send a request to the service in the VM, triggering an access to the resource. Hence, the resource page will be among the pages accessed during processing of the request. Yet, we will also record other pages, such as the ones containing code required to execute the service or accessed to perform other concurrent activities in the VM.

To identify the resource within this set of pages, we use an iterative approach. This allows us to adapt the number of iterations depending on the amount of noise by repeating the following steps $n \in \mathbb{N}$ times. Additionally, we define the current iteration of our identification process as i, with $1 \leq i \leq n$. We start each iteration i by requesting the resource via the service and recording all pages accessed by the VM while fulfilling the request:

$$RQ_i \coloneqq \{p \in P : \text{Page } p \text{ accessed during request } i \text{ for target resource}\}$$

Since we trigger the access to the resource, our sample set is guaranteed to contain the resource pages:

$$R \subseteq RQ_i$$

RQ_i is typically large ($|R| \ll |RQ_i|$), as it not only contains the resource page, but also other pages, such as pages containing code for the service and other processes. We consider all of these other pages as noise, as they directly increase $|RQ_i|$. To reduce this noise, we make use of the fact that each $R \subseteq RQ_i$ and calculate the set of pages RS_i accessed for every single request. We update this set in each round by intersecting RQ_i from the previous set RS_{i-1}:

$$RS_i := RS_{i-1} \cap RQ_i \qquad RS_0 = RQ_1$$

By updating RS_i after each iteration, we are able to filter noise such as pages access from other processes. In other words, after an appropriate number of iterations, RS_i should only contain pages required to process our request.

After this step, the set is smaller, but will still contain pages that are required to fulfill the request, but are not part of the resource, such as code pages from the service. To remove these pages from the set, we track the VM's page accesses while performing a similar request without accessing the resource. An example for such a request could be a different web page from the same service. Again, we track and record all pages accessed during the request:

$$NRQ_i := \{p \in P : \text{Page } p \text{ accessed during request } i \text{ for different resource}\}$$

As we do not trigger the access to the resource for this request, NRQ_i only contains pages from the target resource if another client accesses the resource during our tracking operation. Hence, we assume that pages in NRQ_i are unlikely to contain $p \in R$. Based on this assumption, we define a set of likely candidates RC_i for each round. For this set, we subtract the pages accessed for a request to a different resource NRQ_i from the set RS_i containing only pages accessed for every request:

$$RC_i := RS_i \setminus NRQ_i$$

This method allows us to filter pages that are part of the service, but do not represent the resource itself. Next, we gather the candidates from all iterations and their frequency of occurrence. For this, we make use of multisets, which can contain multiple instances of each element. In order to differentiate the multiset from the traditional set, we indicate the multiplicity of each element:

$$M := \{p_1^2, p_2^1\}$$

To denote the multiplicity of an element p in a multiset M, we use a function from multiset to integer [104]:

$$M(p) = \{ \text{ Multiplicity of } p \text{ in } M\}$$

Hence, taken the multiset

$$M = \{p_1^2, p_2^1\}$$

the multiplicity of the different elements are

$$M(p_1) = 2$$
$$M(p_2) = 1$$
$$M(p_3) = 0$$

Furthermore, we denote the cardinality of a multiset M with:

$$|M| = \sum_{p \in M} M(p)$$

Therefore, taken again the multiset

$$M = \{p_1^2, p_2^1\}$$

we calculate its cardinality with

$$|M| = M(p_1) + M(p_2) = 2 + 1 = 3$$

Next, we define the following operations. As the union of a multiset M with a set S, we specify the result to be a multiset in which the multiplicity of each element p in the original multiset is increased by one if p is also contained in the set. Additionally, p is added to the resulting multiset if it is only contained in S. If p is only contained in M but not in S, its multiplicity remains the same:

$$(M \uplus S)(p) := \begin{cases} M(p) + 1 : p \in S \\ M(p) : p \notin S \end{cases}$$

Additionally, we specify the result of an intersection between a multiset M and a set S to be a multiset in which the multiplicity of each element p is the same as in the original multiset if p is also contained in the set. Otherwise, we remove p from the resulting multiset.

$$(M \cap S)(p) := \begin{cases} M(p) : p \in S \\ \emptyset : p \notin S \end{cases}$$

Hence, this operation results in elements being intersected when they are not in the set S. In comparison, when an element appears in the set, its multiplicity will remain unchanged.

Having established these operations with sets and multisets, we define the multiset of overall candidates OC_i, which provides information about how often we have identified each page as candidate for the target resource. Following our specifications, we define OC_i as:

$$OC_i := (OC_{i-1} \uplus RC_i) \cap RS_i \qquad OC_0 = \emptyset$$

With the first term of the equation, $(OC_{i-1} \uplus RC_i)$, we ensure that all candidate pages are represented in OC_i according to the number of their occurrences as round candidate. In other words, a page occurring as likely candidate in three rounds would have a

multiplicity of three. By additionally intersecting the set with RS_i, we remove candidates from previous iterations ($p \in OC_{i-1}$) that we can exclude with the knowledge gained in the current iteration ($p \notin RS_i$).

Finally, we calculate for each candidate page $p \in OC_i$ the probability of it being part of the target resource based on how often it was a candidate:

$$\mathrm{P}_i[p \in R] := \frac{OC_i(p)}{|OC_i|}$$

Note that if $|R| > 1$, the probability is distributed between all $p \in R$. Therefore, we only interpret the probability in relation to the probability of other pages $p \in OC_i$. Using this approach, we calculate the probability $\mathrm{P}_n[p \in R]$ for each $p \in OC_n$ after every iteration, and build a list of candidate pages sorted by probability.

To further clarify our approach, let us go through a simplified example. Let us assume that during our first iteration ($i = 1$), we record the following page accesses RQ_1 when requesting the target resource:

$$RQ_1 = \{4, 8, 15, 16, 23, 42\}$$

Additionally, we record the page accesses NRQ_1 when requesting a different resource:

$$NRQ_1 = \{3, 8, 12, 15, 16, 23, 27\}$$

Using this information, we first calculate RS_1, which equals RQ_1 in the first iteration.

$$RS_1 = RS_0 \cap RQ_1 \qquad RS_0 = RQ_1$$
$$RS_1 = \{4, 8, 15, 16, 23, 42\} \cap \{4, 8, 15, 16, 23, 42\} = \{4, 8, 15, 16, 23, 42\}$$

Next, we calculate the list of likely candidates for the first round RC_1 by subtracting NRQ_1 from RS_1. This allows us to eliminate all pages from the set which are unlikely to contain the target resource:

$$RC_1 = RS_1 \setminus NRQ_1 = \{4, 42\}$$

In the last step of the iteration, we add the set of this round's candidate pages RC_1 to the multiset of overall candidates OC_0. By using a multiset, we can determine the number of iterations in which each page has been a candidate. We then intersect this multiset with the set RS_1 of pages accessed to fulfill the first request to the resource. This enables us to remove candidates which were not contained in RS_1 and therefore cannot represent the target resource:

$$\begin{aligned} OC_1 &= (OC_0 \uplus RC_1) \cap RS_1 = (\emptyset \uplus \{4, 42\}) \cap \{4, 8, 15, 16, 23, 42\} \\ &= \{4^1, 42^1\} \cap \{4, 8, 15, 16, 23, 42\} = \{4^1, 42^1\} \end{aligned}$$

Having determined the set of overall candidates after the first round OC_1, we calculate the probability of each page $p \in OC_1$ being part of the resource. For this calculation, we compare the multiplicity of the different elements in the multiset OC_1 by dividing their multiplicity by the total number of elements in OC_1, which we defined as $|OC_1|$. Note that after the first round, all pages recorded in the set of overall candidates OC will have the same multiplicity, namely one. Hence, we determine the same probability for each page $p \in OC_1$ to be part of the resource.

$$\mathrm{P}_1[p \in R] = \frac{OC_1(p)}{|OC_1|}$$
$$\mathrm{P}_1[4 \in R] = \frac{1}{2} = 0.5$$
$$\mathrm{P}_1[42 \in R] = \frac{1}{2} = 0.5$$

During the second iteration ($i = 2$), we record:

$$RQ_2 = \{6, 8, 15, 16, 23, 42\}$$
$$NRQ_2 = \{2, 8, 12, 13, 15, 23\}$$

With this information, we calculate RS_2, RC_2 and OC_2:

$$\begin{aligned} RS_2 &= RS_1 \cap RQ_2 = \{4, 8, 15, 16, 23, 42\} \cap \{6, 8, 15, 16, 23, 42\} = \{8, 15, 16, 23, 42\} \\ RC_2 &= RS_2 \setminus NRQ_2 = \{8, 15, 16, 23, 42\} \setminus \{2, 8, 12, 13, 15, 23\} = \{16, 42\} \\ OC_2 &= (OC_1 \uplus RC_2) \cap RS_2 = (\{4^1, 42^1\} \uplus \{16, 42\}) \cap \{6, 8, 15, 16, 23, 42\} \\ &= \{4^1, 16^1, 42^2\} \cap \{8, 15, 16, 23, 42\} = \{16^1, 42^2\} \end{aligned}$$

To finish the round, we calculate the possibility for each $p \in OC_2$ to be part of the resource. As the different elements now have different multiplicities, we also determine different probabilities for each page:

$$\mathrm{P}_2[p \in R] = \frac{OC_2(p)}{|OC_2|}$$
$$\mathrm{P}_2[16 \in R] = \frac{1}{3} = 0.33$$
$$\mathrm{P}_2[42 \in R] = \frac{2}{3} = 0.67$$

Hence, after two rounds, we determine page 42 to be the most likely candidate to represent the resource. This example shows that our method enables us to remove noise during the sampling phase, allowing us to identify the resource using only a small number of iterations.

4.2.3 Secret Identification

Having identified the resource, we could simply remap the GFN containing the resource to an arbitrary SFN. By afterwards requesting the resource from the service, we would be able to retrieve the unencrypted content of the respective page. However, the chances of this arbitrary page containing valuable information are limited. At the same time, retrieving all of the VM's pages to identify all secrets would require a high number of requests. Let us assume that the VM's memory comprises 2 GB, which equals 2,097,152 KB, and is divided into pages of 4 KB. If the resource spans exactly one page, we would need $2{,}097{,}152/4 = 524{,}288$ requests to extract all of the VM's pages.

To avoid having to extract such a large number of pages, we instead identify secrets within the VM's memory (❷), which we then extract in a targeted fashion. For this identification, we again make use of *AV1: Page Access Tracking*, in particular page access tracking. Additionally, we combine this attack vector with *AV2: Network Monitoring* and *AV3: Disk Access Monitoring*. To be precise, we track all of the VM's pages until we record an activity indicating that the VM has recently accessed a secret. We will use *AV2* and *AV3* to identify such an activity. Tracking all accessed pages until the activity allows us to limit the set of pages which can contain the secret to the list of pages tracked before the activity.

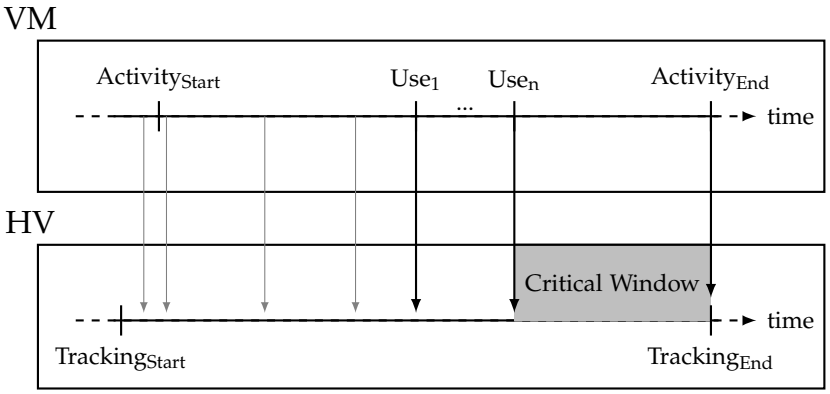

Figure 4.2: We identify the secret within the VM's memory by performing page tracking during an activity for which the VM accesses the secret. To ensure that the page containing the secret is among the tracked pages, we should avoid to start tracking in the critical window.

Let us clarify our approach with an example in which we aim to extract a TLS private key from the VM. To identify the private key within the VM's memory, we activate page access tracking on all of the VM's pages. Next, we wait for an activity indicating that the VM has recently accessed the private key. In case of the TLS private key, such an activity can be a successful TLS handshake. The activity of a TLS handshake fulfills two important criteria. For one, it requires the VM to access the secret, in this case the private key, to prove its identity. Furthermore, the hypervisor can detect the end of the

activity, which is the final message in the handshake, with *AV2: Network Monitoring*. Another example for a secret in the VM's memory would a FDE key. The respective activity would be an access to the encrypted disk, which we can observe with *AV3: Disk Access Monitoring*.

Figure 4.2 shows the detailed identification process. While the upper rectangle depicts the VM, the lower rectangle depicts the hypervisor. Both the VM and the hypervisor have timelines running in parallel, indicating events in the respective entity. We begin our secret identification with the observation phase, in which we track accesses to all of the VM's pages. We denote the start of this phase with $Tracking_{Start}$. With the access tracking, the VM traps into the hypervisor on every page access, depicted by the vertical arrows.

At some point after $Tracking_{Start}$, the VM will start the activity requiring it to access the secret, denoted by $Activity_{Start}$. During the activity, the VM accesses the secret once or multiple times, denoted by Use_i to Use_n. Due to the page access tracking, we record the access to the secret. Yet, we are not able to differentiate these specific accesses from other page accesses.

After a certain amount of page accesses, the VM finishes its activity, denoted by $Activity_{End}$. Having chosen an activity with an observable end, we then stop the tracking process, denoted by $Tracking_{End}$. At this point, we can be sure that at least one of the tracked pages contains the secret. In other words, this approach permits us to significantly reduce the number of candidate pages possibly containing the secret.

Yet, we have to keep in mind that we start the tracking at an arbitrary point in time and that $Activity_{Start}$ might not be observable from the hypervisor. Hence, it is possible that we start tracking between the last access to the secret, Use_n, and $Activity_{End}$. This would cause the list of tracked pages to not contain the page holding the secret. For this reason, we define the time slot between Use_n and $Tracking_{End}$ as the *critical window*. In case we start the tracking process in this critical window, it would have to be repeated. However, our experience shows that the chances to start tracking exactly in this slot are limited. Furthermore, even repeating the targeted extraction multiple times will be more efficient than extracting the VM's entire memory.

To sum up, to perform the secret identification (❷), we observe the VM's behavior while it accesses the secret to be extracted. This provides us with a list of candidate pages possibly containing the secret, which is significantly smaller than the complete set of pages. Although this approach does not identify the exact page containing the secret, it allows us to drastically reduce the set of candidates.

4.2.4 Secret Extraction

Having gathered the list of pages the VM accessed between $Activity_{Start}$ and $Activity_{End}$, we continue with the secret extraction (❸). In this phase, we first establish the sequence in which we extract the candidate pages determined in the previous step. The goal of this sequence is to sort the list of tracked pages according to the probability that they contain the secret. For this, we make use of our knowledge that $Activity_{End}$ indicates that

the VM has recently accessed the secret. In other words, pages tracked shortly before $Activity_{End}$ have a higher chance to contain the secret. Therefore, we reverse the order of tracked pages, thereby extracting the page accessed just before $Activity_{End}$ with the highest priority.

The possibilities to further filter and prioritize the list or tracked pages depend on the secret to be extracted. Sticking to our example of the secret being a TLS private key, we can assume that the VM performed a read operation on the key. Hence, we can eliminate all execute accesses from the list, as the secret will not be stored in executable memory. Yet, there is the possibility that the VM accessed the same page for a write operation before reading the secret, and only afterwards performed the read operation. In such a scenario, we would only record the write access, as we only track the first access to each page (Section 4.2.1). In order not to exclude such scenarios in the search phase, we instead only reduce the priority of pages with recorded write accesses, and prioritize pages from which the VM read data.

Having created a probability list of pages holding the resource (Section 4.2.2) and the sequence in which to extract the candidate pages possibly containing the secret (Section 4.2.3), we start the extraction process. For this extraction, we make use of our most important attack vector, *AV4: SLAT Remapping*. *AV4* is the hypervisor's ability to remap a GFN to a different SFN. To successfully exploit this attack vector, we first determine the number of pages r that are at least necessary to store the target resource. We then take the first r pages of the probability list and repeat the following three steps:

1. **Page Remapping.** We modify the SLAT entries of the r pages so that their GFNs point to the first r candidate pages (❸). After the modification, we flush the corresponding Translation Lookaside Buffer (TLB) entries to ensure the changes take effect immediately.

2. **Data Request.** We request the target resource from our service. Since we remapped the underlying pages for the resource, the service will unintentionally respond with data from the candidate pages.

3. **Data Analysis.** We analyze the data contained in the response regarding whether in contains the secret to be extracted. If this is not the case, we go back to the first step, and remap the GFNs to the next r pages of the candidate list.

If we receive the original resource for one or more of the r pages replaced in the first step, the respective pages do not belong to the target resource. In this case we continue the extraction with the next pages in the probability list from Section 4.2.3.

4.2.5 Evaluation

Having described the three different steps of our attack, we perform a detailed evaluation in which we measure the performance and precision of all three steps in different scenarios and with varying noise.

The SEVered Framework

For the practical evaluation of *RQ1: Which attack vectors enable attacks against a TEE's cryptographic memory isolation without memory integrity?*, we created the SEVered framework[3], our contribution *C3*. The SEVered framework is an extension to Linux' virtualization solution Kernel-based Virtual Machine (KVM) [80] that enables an attacker in control of the hypervisor to exploit various attack vectors described in this chapter. We use this framework for the evaluation of our attack against the VM's confidentiality and in Section 4.3.4 for the evaluation of the attack against the VM's integrity.

For the evaluation of our attack against the VM's confidentiality, our framework provides the ability to perform access and execution tracking additionally to the already existing write tracking [56]. Additionally, it includes functionality that allows to remap GFNs to different SFNs by altering the SLAT.

We made our framework publicly available for it to be used not only for our analysis of AMD SEV, but also of other TEEs protecting VMs, such as Intel TDX [69], IBM PEF [64] and ARM CCA [8]. While to the best of our knowledge these technologies do provide effective memory isolation, their resistance against attack vectors such as *AV4: SLAT Remapping* can be verified with the help of our framework. Furthermore, it permits to exploit other attack vectors such as *AV1: Page Access Tracking* on these TEEs, which could be useful in combination with other attack vectors not covered in this work.

Setup

We implemented our prototype on a system powered by an AMD EPYC 7251 processor with SEV enabled. Our system ran Debian Linux with the SEV-enabled kernel in version `4.13.0-rc1` and QEMU `2.9.50`, as provided by AMD[4]. Additionally, we patched the kernel with our SEVered framework to be able to exploit the different attack vectors.

Our target VM used FDE with the AES-XTS algorithm and a 256 bit key. Within the VM, we were running two web servers, Apache `2.4.25-3` and nginx `1.10.3-1`, as well as an SSH server, OpenSSH `1:7.4p1-10`. As target resource to be served by these services, we used a 4 KB file spanning exactly one memory page. Additionally, we introduced different *noise levels* into our target VM, allowing us to evaluate each step in a real-world scenario. A noise level of 20 refers to an environment where on average 20 random accesses per second are made to our three services by arbitrary clients.

Resource Identification Evaluation

Using our setup, we first evaluated our resource identification approach (❶) exploiting *AV1: Page Access Tracking*. An important factor for the tracking is the noise model that requests different resources from the VM's services in order to simulate concurrent activity by other clients. We call this RQ-noise, as it directly influences $|RQ_i|$, the number

[3] `https://github.com/Fraunhofer-AISEC/severed-framework`
[4] `https://github.com/AMDESE/AMDSEV`

of pages we record when requesting the resource (Section 4.2.2). Furthermore, the noise model also generates requests to the resource which we aimed to identify. These requests are an important factor for our evaluation, as a concurrent access to the resource while performing the request NRQ_i to a different resource would cause us to track an access to the target page for this request. Hence, we call this noise NRQ-noise.

Having the impact of noise in mind, we conducted eight test runs with 100 iterations on four different noise levels for each of the three services offered by the VM. In each iteration, we first determined the page containing the resource. Based on this knowledge, we recorded the set of pages our algorithm identified to as likely contain the resource as the actual page containing the resource. In other words, the smaller this set, the better the identification.

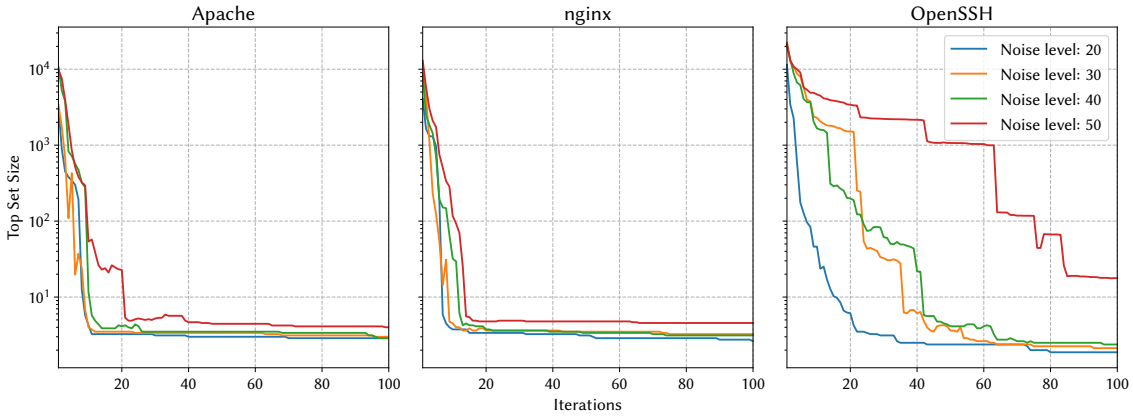

Figure 4.3: Measurements of top set size $|T|$ for increasing number of iterations and different noise levels. While the X-axes depict the number of iterations, the Y-axes indicate the respective size of the top set. Source: [107].

Figure 4.3 shows the average size of the set for each service, noise level, and iteration. We use a logarithmic scale for the Y-axis, as the top set size quickly decreases after a few iterations. The graphs on the left and in the middle depict that for both Apache and nginx, the top set size quickly converges to about 3 to 4 candidates one average for all noise levels. Similarly, also with OpenSSH the top set size converges to an average of about 2 to 3 candidates with most levels. Only with noise level 50 in OpenSSH we were not able to identify a small top set within 100 iterations.

These results show that we can apply our identification mechanism to noisy environments by dynamically increasing the number of iterations. Table 4.1 summarizes the number of iterations and time required for every service and noise level until the top set converges to a size smaller or equal to 5 candidates. For the web servers, the top set converged in less than ten iterations and ten seconds for the lower noise levels. Even with the highest noise level, this took less than 23 seconds and required at most 22 iterations. In comparison, for OpenSSH we required between 21 and 46 iterations and 38.85 to 111.09 seconds for the lower noise levels. Only for the highest noise level we were not able to reduce the set to 5 or less candidates within 100 iterations.

Table 4.1: Number of iterations and time required until top set converges to a size smaller or equal than 5 for different noise levels. Source: [107].

Noise Level	Apache	nginx	OpenSSH
20	10 (7.4 s)	8 (5.56 s)	21 (38.85 s)
30	10 (7.5 s)	9 (6.62 s)	42 (85.47 s)
40	12 (9.7 s)	13 (13.2 s)	46 (111.09 s)
50	22 (23.0 s)	16 (17.84 s)	>100 (>5 min)

Table 4.2: NRQ-noise probability and average recording size for different noise levels and scenarios. Source: [107].

Noise Level	Apache	nginx	OpenSSH
20	35 % (8,220)	34 % (8,355)	63 % (18,960)
30	49 % (10,860)	51 % (10,360)	78 % (21,475)
40	60 % (13,040)	62 % (12,430)	85 % (23,015)
50	74 % (15,950)	69 % (15,970)	90 % (24,990)

To confirm that our noise model works as intended, we additionally analyzed the results regarding the contained noise. In particular NRQ-noise is critical, as it removes the target page from the candidate list of the respective iteration. In Table 4.2, we present the probability of an iteration being influenced by NRQ-noise. The table also shows the average recording size. These measurements reveal that our noise model introduces significant NRQ-noise in up to 90% of the NRQ_i recordings and strongly increases their size. Nevertheless, we were able to quickly reduce the number of candidates to 5 or less for most scenarios and noise levels.

Among the candidate pages, we always identified the last commonly accessed page to contain the target resource for all services. This observation is comprehensible, as all services must perform operations, such as opening a socket, before transmitting the requested content. Together with the small number of candidates, those observations allowed us to correctly identify the page of our target resource in all our test cases. These results show that our identification mechanism enables us to quickly identify the resource in the VM's memory. Furthermore, the mechanism is robust against noise and thus applicable in real-world scenarios.

Secret Identification Evaluation

Having identified the resource (❶), our next step is the identification of secrets in the VM's memory (❷). We evaluate the secret identification for two different types of keys: private keys of different services and FDE keys.

To identify the private key of a service, we make use of *AV2: Network Monitoring* and the fact that services such as a web server or SSH server use the keys to prove their

identity during a handshake. This enables us to define $Activity_{Start}$ as the start of the handshake. Therefore, Use_i represents the service using the key during the handshake. The exact point in time of Use_i depends on the key exchange method. For example, considering an ECDHE-based key exchange algorithm, the server will access the private key to sign the curve parameters [15]. In comparison, during an RSA-based key exchange, the server accesses the key to decrypt the premaster secret encrypted with the server's public key [40]. In any case, we are not able to determine when Use_i occurs exactly. Yet, we do know that it will occur between $Activity_{Start}$ and the end of the handshake. Hence, we decided to use the `change cipher spec` packet as indicator for $Activity_{End}$. This packet initiates the switch to a different cipher, indicating that the cipher setup has been completed successfully. While we could also use other packets sent earlier during the handshake for $Activity_{End}$, the `change cipher spec` packet is independent from the encryption algorithm. Also note that in the private key scenario, we could observe, or even trigger $Activity_{Start}$.

Another valuable secret in the VM's memory are FDE keys. Such keys are required as SEV does not protect the virtual hard disk. Therefore, the VM needs to additionally encrypt its disk to protect it from a malicious hypervisor. The respective encryption key will remain in the VM's memory, as it is required for all I/O disk operations. Those are the two properties on which we capitalize when identifying the FDE key. To be precise, we define $Activity_{Start}$ as the start of an I/O disk operation requiring access to the key. Additionally, we again define the access to the secret itself with Use_i. What remains is the definition of an $Activity_{End}$ which we can observe from the hypervisor. For this, we make use *AV3: Disk Access Monitoring*, the fact that the hypervisor is able to monitor accesses to the VM's virtual disk. Specifically, we monitor the VM's disk image, and wait for an `inotify` event. This event indicates a successful I/O operation on the VM's disk, serving us as $Activity_{End}$ for the identification of the FDE key.

In order to evaluate the precision of our method for both private keys and FDE keys, we evaluated four different scenarios: extraction of the private key from Apache, nginx and OpenSSH, and extraction of the VM's FDE key. To observe the handshake messages of the services, we made use of `tcpdump` in combination with `libpcap`. We patched `libpcap` to execute a system call that starts access tracking at the same time as packet capturing. Additionally, we added a call to stop tracking as soon as we captured the packet indicating $Activity_{End}$. For the two TLS servers, this packet was the `change cipher spec` packet. In comparison, we used the `new keys` packet for OpenSSH, which concludes the SSH handshake. Furthermore, we made use of the `inotifywait` tool to observe inotify events, which allowed us to detect disk writes on the VM's disk image. We also patched `inotifywait` to start tracking before watching events and to stop tracking as soon as an event was observed. By patching both `libpcap` and `inotifywait`, we were able to minimize the time between $Activity_{End}$ and $Tracking_{End}$.

The activities were triggered by the generated noise, which requested a resource from either Apache or nginx with probability of $\frac{300}{301}$. With a probability of $\frac{1}{301}$, it initiated an SSH login and kept the session running for two minutes. This distribution simulates a

real-world scenario, in which SSH traffic will occur less frequently. With this configuration, the average duration before we observe $Activity_{End}$ for the web servers lies between a few hundreds of milliseconds and a few seconds, depending on the noise level. In comparison, it may take anywhere between tens of seconds and a few minutes before we can observe an SSH handshake and conclude the secret identification phase for the SSH private key.

Using this setup, we performed 2,000 independent iterations of the secret identification process for each scenario and four different noise levels. While the VM was processing the incoming requests caused by the noise, we started the access tracking at a random point in time. After the tracking, we made use of our knowledge that for all four scenarios, the targeted secrets were read or written by the VM, and therefore located in non-executable memory. Hence, we performed a postprocessing step in which we filtered execute-accesses from the list of candidate pages.

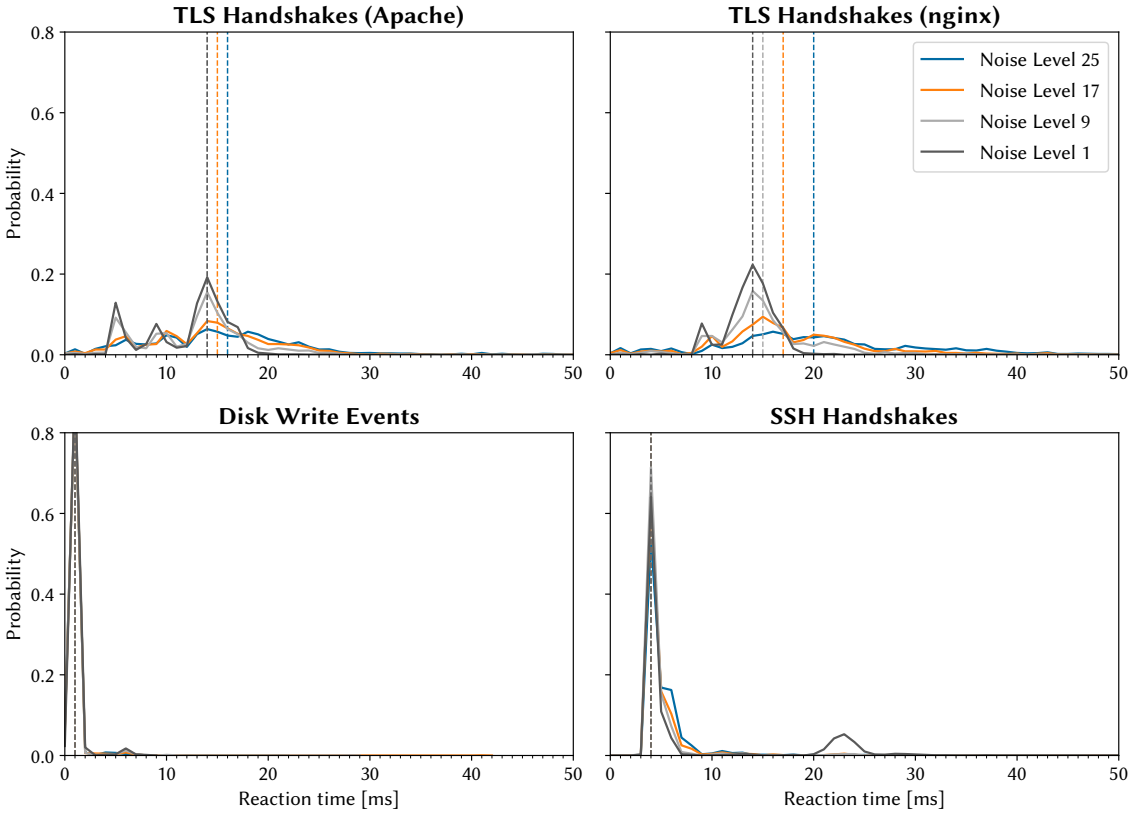

Figure 4.4: Distribution of the reaction times over different noise levels for different scenarios. While the X-axes depict discretized time steps of one millisecond, the Y-axes are normalized to one. Source: [106].

Having performed the measurements, we analyzed the time we require to stop the tracking after the VM's last access to the secret, Use_n. We call this the *reaction time*. The smaller the reaction time, the lower the chance of tracking additional pages after Use_n.

In other words, a smaller reaction time improves the accuracy of our secret identification. The reaction time consists of two parts. The first part is the critical window, which is the time between Use_n and $Activity_{End}$ (Section 4.2.3). As we are required to wait until the detection of $Activity_{End}$ to stop tracking, we are no able to influence this part of the reaction time. However, the second part of the reaction time is the time we require to stop tracking after we detect $Activity_{End}$. We are able to reduce this part by stopping the tracking as soon as we detect $Activity_{End}$.

Figure 4.4 depicts the reaction times for our four different scenarios. In the figure, the X-axes represent the reaction time in milliseconds. Furthermore, the Y-axes depict the probability of a reaction time in the respective time slot, discretized to one millisecond. In each figure, the graphs represent the different noise levels, 1, 9, 17 and 25. Additionally, the dashed vertical lines show the median reaction time for each noise level.

The diagrams on the top represent the results for the Transport Layer Security (TLS) handshakes of Apache and nginx. Both indicate a reliable reaction time for the lower noise levels by showing a clear peak. To be precise, for noise level 1, the reaction time never exceeded 21 milliseconds for Apache or 22 milliseconds for nginx. However, with an increased noise level, in which the VM executes a higher amount of concurrent activities, the measurements are more scattered over time. Specifically, we measured a reaction time of up to around 50 milliseconds in rare cases for both Apache and nginx. Yet, the median reaction time increased to only about 16 milliseconds for Apache, and 20 for nginx. This higher median confirms that the point at which the VM executes Use_n becomes more scattered with higher noise. Furthermore, the higher median indicates that the VM performs a higher number of concurrent activities with higher noise and hence also accesses a higher number of pages.

The diagram on the bottom left in Figure 4.4 shows that for disk write events, our implementation achieved a very fast reaction time of one millisecond in the median for all noise levels. In only a few cases, we measured a slightly higher reaction time. Another interesting observation is that in contrast to the web server scenarios, the reaction time was generally independent from the noise level. We attribute this to the fact that intercepting disk write events is less complex than capturing network packets, and therefore introduces less delay.

The diagram on the bottom right shows that for SSH handshakes, we measured a median reaction time of around four milliseconds. Similar to the disk write events, the measurements were mostly independent from the noise level. In only a few cases, we encountered a reaction time of about 30 milliseconds.

To further quantify the precision of our approach, we measured the time we required for the secret identification as well as the median number of tracked pages, depicted in Table 4.3. For both web servers, the identification time, which is the time between $Tracking_{Start}$ and $Tracking_{End}$, was between 0.31 and 1.46 seconds. Notably, a higher noise level decreased the identification time, as we had to wait on average less time before recording $Activity_{End}$. The shorter tracking time also decreased the median number of tracked pages, which we measured to be between 1,691 and 2,085.

Table 4.3: Time required for the secret identification process for four different scenarios and noise levels, including the median number of tracked pages.

Scenario	Noise Level	Identification Time [s]	Median Number of Tracked Pages
TLS (nginx)	1	1.46	2,085
	9	0.37	1,954
	17	0.32	1,944
	25	0.31	1,965
TLS (Apache)	1	1.42	1,975
	9	0.37	1,707
	17	0.33	1,717
	25	0.32	1,691
FDE	1	2.43	2,525
	9	2.15	3,225
	17	2.08	3,432
	25	2.04	3,669
SSH	1	193.36	11,093
	9	27.23	10,102
	17	16.19	10,473
	25	14.41	10,846

In comparison, the time required for the identification in the FDE scenario only decreased slightly with higher noise levels. Therefore, also the median number of tracked pages did not decrease, but increased from 2,525 with noise level 1 to 3,699 with level 25.

For SSH, identification on the lowest noise level took the longest, namely 193.36 seconds. This is caused by the fact that our noise generation only rarely triggers SSH handshakes. Yet, similar to the web servers, the identification time decreased with the higher noise levels to as little as 14.41 seconds. In comparison, the median number of pages we tracked remained relatively stable between 10,846 and 11,093.

It is important to note that the median number of tracked pages only indicates the number of pages we have to extract in the worst case, assuming that we did not start tracking within the critical window. However, a smart prioritization of the extraction order will permit us to further reduce the number of pages we have to extract before being able to determine the secret (Section 4.2.4).

Secret Extraction Evaluation

Having evaluated both the resource and the secret identification, we continue with the evaluation of the secret extraction. In this phase, we iteratively extract pages from the

sequence of pages identified during the secret identification and search the content of the respective page for the secret. To prioritize the sequence, we extract the pages in the reverse order of which we tracked them (Section 4.2.4).

When searching an extracted page for a secret, we make use of its publicly known information. Specifically, for the web and SSH servers, we know the server's public key and its length. Using for example RSA, the private components are the factors p and q of known length, which divide the modulus of the public key. Using this knowledge, we can traverse every chunk of extracted memory and analyze if it contains a contiguous bit sequence which divides the modulus without remainder. If this is the case, we have successfully identified either p or q, and are able to also calculate the other factor.

In comparison, when searching the page for the FDE key, we make use of the fact that the VM's kernel stores the key in a specific data structure. In this structure, various fields are filled with values in a certain range, for example with kernel addresses pointing to other kernel objects. These characteristics allow us to identify the structure in the VM's memory. To further increase the possibility for key identification, we capitalize on the fact that different data in the VM's memory can enable us to retrieve the FDE key. Specifically, FDE keys are usually based on AES. Using the AES algorithm, the kernel derives round keys and keeps them in memory in so-called *AES key schedules*. Hence, we can additionally use the statistical properties of such key schedules to identify the FDE key [58]. Having identified the FDE key either via the AES key schedules or via the key data structure, we can verify the key by attempting to decrypt the VM's disk image.

To evaluate the secret extraction, we extracted the private keys from all three services in the VM, Apache, nginx and OpenSSH, which all had a length of 4,096 bit. Additionally, we extracted the 256 bit AES-XTS FDE key.

For the memory extraction itself, we used a resource from nginx identified during the resource identification (Section 4.2.2) We then iteratively remapped the resource to all pages determined during the secret identification (Section 4.2.3) and extracted the page by requesting the resource from nginx (❸). In case the targeted secret was located on the extracted page, we stopped the extraction process. Otherwise, we continued by extracting and analyzing the next page in the sequence.

Table 4.4 gives an overview of our evaluation results. For each scenario and noise level, we measured the time required for the extraction. Additionally, we determined the median number of pages we had to extract to identify the secret and the mean absolute deviation from this number.

In both web server scenarios, higher noise had an impact on the extraction time, which ranged from 15.48 to 32.71 seconds. For nginx, we had to extract in the median 102 pages on noise level 1 and 301 pages on level 25 to identify the secret. In other words, we had to extract between 408 KB and 1,024 KB of the VM's memory before we were able to determine the private key. In comparison, for Apache, we required a slightly higher median of 128 pages, or 512 KB, to extract the private key on noise level 1. Yet, this number only increased to 171 pages, or 684 KB, on noise level 25. Along with the median page number increasing for both web servers, also the mean absolute deviation

Table 4.4: Time required for the secret extraction for four different scenarios and noise levels, including the median number of extracted pages and the mean absolute deviation.

Scenario	Noise Level	Extraction Time [s]	Median Number of Extracted Pages	Mean Absolute Deviation
TLS (nginx)	1	17.72	102	5
	9	15.48	116	19
	17	18.61	165	69
	25	32.71	301	160
TLS (Apache)	1	21.90	128	21
	9	17.95	137	40
	17	17.44	154	80
	25	18.65	171	109
FDE	1	12.24	70	8
	9	9.34	71	9
	17	7.84	70	8
	25	7.37	69	9
SSH	1	1.33	7	1
	9	0.97	7	1
	17	0.83	7	1
	25	0.80	7	1

increased with the noise level. Nevertheless, in both scenarios, the number of pages we had to extract was only a fraction of the total number of pages tracked during the secret identification. It is also worth noting that for both web servers, we measured an average extraction time of around 123 milliseconds for a single page, with fluctuations in the scale of a few tens of milliseconds. To following search of the private key within the extracted memory page took around 50 milliseconds.

In comparison, the extraction time in the FDE key scenario decreased with more noise, reducing it from 12.24 to 7.37 seconds. Another difference is that the median number of pages we had to extract remained stable between 69 and 71, corresponding to 276 and 284 KB of memory. As for the web servers scenarios, this numbers represent only a fraction of the total number of pages tracked during the secret identification. Similar to the median number of extracted pages, also the mean absolute deviation from the median stayed stable between 8 and 9. Lastly, searching an extracted page for an FDE key only took 2 milliseconds. Most of the times, we identified the key as part of the AES key schedule, and only only in rare cases by the kernel data structure.

Extracting the private key from the SSH server took between 1.33 seconds with noise level 1, and 0.80 seconds with level 25. Also the median number of pages we extracted was not only very small, but additionally remained completely stable at 7. Therefore, also the mean absolute deviation remained constant at 1. These strong results are especially important in the SSH scenario, as `sshd` forks a new process for every SSH connection. As soon as the process is terminated, the private key will be deleted from memory. In order words, we have to conclude the secret extraction before the SSH connection is closed. However, our very fast extraction time of maximum 1.33 seconds facilitates achieving this goal. Similar to the web servers, searching an extracted memory page for the SSH server's private key took around 50 milliseconds.

Another interesting observation is that we were able to extract the secret in 99.99 % of all test runs for both web servers and the FDE key scenario. In the SSH scenario, we measured a comparability high success rate of 99.98 %. These results indicate that in the previous secret identification phase, we started the tracking within the critical window in only a few cases. Also note that if we start tracking within the critical window, there is still the possibility of another handshake being processed concurrently and triggering an access to the secret. This would allow us to determine the private key even when starting tracking in the critical window of the original handshake. This applies especially to higher noise levels, in which a high number of handshakes is performed each second. Hence, the critical window can be smaller especially on higher noise levels.

In summary, our evaluation shows that we were able to identify the resource in less than 22 iterations for both Apache and nginx on different noise levels (Section 4.2.5). Additionally, we identified the resource provided by OpenSSH in 46 iterations or less on most noise levels. Having identified the resource, we were able to drastically reduce the number of pages possibly containing the secret during the secret identification (Section 4.2.5) with a very high success rate. In the following secret extraction phase, we started by extracting the most promising candidate pages. This only required us to extract kilobytes of memory from the VM's memory, allowing us to retrieve the respective secret within seconds.

4.2.6 Discussion

Our evaluation shows that we are able to quickly extract secrets from the memory of SEV-protected VMs. This confirms that our methods can handle varying amounts of noise during both the resource and secret identification. In this section, we discuss our evaluation results regarding their overhead, behavior of the VM, possibilities for reduced attack times and their general applicability.

Overhead To identify both the resource and the secret, we make use of *AV1: Page Access Tracking*. Our page tracking mechanisms causes the VM to trap into to the hypervisor when attempting to access a tracked page. This additional step will increase the VM's overall processing times. However, as our page tracking only triggers a single page fault for every tracked page, this increase is limited. To be precise, we did not detect any

noticeable delays in neither the hypervisor nor the VM. Instead, both the hypervisor and the VM remained stable and responsive even on the highest noise levels. The only effect from our tracking we were able to detect was a small additional delay in the response time of both the web servers and the SSH server.

Low Memory In case the VM is low on memory, its kernel might try to free memory. For this process, the kernel might try to swap out pages, kill processes, or unmap file-backed pages. Especially the latter is critical for our attack, as it could cause the page originally containing the secret to be re-used by another process or the kernel itself. Therefore, although we would still be able to extract the respective page's memory contents, the secret may have already been overwritten. Yet, we did not encounter such behavior during our tests.

Triggering Activities During the secret identification phase, we start tracking at an arbitrary point in time. Additionally, we refrain from actively triggering the VM to access the secret in order to interfere as little as possible with the VM's operations. Instead, our noise model caused the VM to regularly access the targeted secret. However, in a real-world scenario, the VM may only rarely access the secret. This could be, for example, the case in the SSH scenario, where the administrator may only use the service from time to time. To avoid having to wait for such an interaction, we could also trigger the VM's access to the secret. Sticking to the example of the SSH server, we could start a login procedure to trigger the server to access its private key. This does not require a valid user account, as the server will already access its private key before the login procedure in order to prove its identity to the client. Hence, the attacker can initiate a login and extract the SSH server's private key before the session will be closed after the default timeout of two minutes. However, note that actively triggering the access to the secret may also decrease the attack's verbosity, and may therefore not always be desired.

Generalization Our approach neither depends on a specific OS running in the VM, nor a specific service, library versions or hardware. Therefore, we expect our approach to be easily transferable to other scenarios and configurations. For example, it can be used to extract confidential code, documents or images. This would only require to slightly adapt the secret identification process (Section 4.2.3) by identifying an activity indicating that the VM recently accessed the respective secret. In case such an activity cannot be identified, it is also possible to skip the secret identification phase. Instead, we could iteratively extract pages from the VM until the analysis of the retrieved memory page reveals the secret.

Furthermore, we expect the performance of our attack to only vary slightly with different hard- or software configurations due to the limited time required to perform the attack. Specifically, we ran several tests in which we assigned additional memory or CPU cores to the VM. Additionally, we tested different numbers of worker processes for

the web servers. In all cases, the performance indicators remained coherent with our evaluation results over all runs.

Prioritizing Specific Pages To further improve the performance of the secret extraction, we could perform additional preprocessing steps between the secret identification and extraction. The details for these steps will depend on the scenario and VM. Considering the TLS handshake scenarios, we determined the private key to be on a read-accessed page in 96 % of all evaluation rounds. In comparison, we discovered the private key from the SSH scenario in 93 % of the rounds on a write-accessed page. Yet, it may depend on other data located on the same page as the secret whether we first track a read- or a write access to the respective page. It is therefore difficult to predetermine the type of access with certainty. Specifically, prioritizing read-accessed pages in the TLS or write-accessed in the SSH scenario may boost the attack in most cases. Yet, it could also produce costly outliers for the few cases in which the page containing the secret was accessed differently than expected.

Utilizing Known Reaction Time We could further optimize the extraction phase by using our detailed knowledge of the reaction times (Figure 4.4). Considering the web servers and the SSH server, we encountered a gap of a few milliseconds between Use_n and $Tracking_{End}$ in most iterations. In other words, it is likely that the secret was not accessed in the last moments before we stopped the tracking. Hence, we could decrease the extraction priority for pages in this time frame. Similarly, the reaction times can also allow to determine the point in time at which to stop the extraction phase in case no secret is found. To be precise, for both web servers and the SSH server, the secret was only rarely accessed more than 30 milliseconds before the end of the tracking. Therefore, if we extracted all tracked pages until this point and were not able to determine the secret, this might be an indication that we started tracking inside the critical window and that it is necessary to restart the secret identification.

4.3 Impact of Missing Memory Integrity on the Integrity of TEEs

In the previous section, we showed how an attacker in control of the hypervisor can exploit SEV's missing memory integrity protection to violate the VM's confidentiality. For the attack, we determined a total of four attack vectors. In this section, we continue by showing that this weakness can also affect the VM's integrity. Specifically, we present our contribution *C2*, an attack that permits to execute arbitrary code within an SEV-protected VM. As with the previous contribution, one of our goals is to highlight the impact when combining the very strong attacker model of a malicious hypervisor with security weaknesses such as missing memory integrity. Furthermore, we want to identify the different attack vectors required to execute the attack in order to systematically discuss countermeasures in Chapter 5.

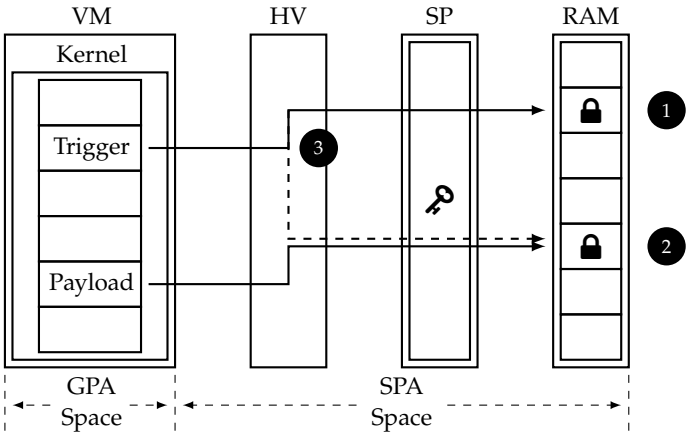

Figure 4.5: Attacking the VM's integrity. Having determined a trigger, we send a payload to the VM. After determining the location of the payload, we modify the SLAT and trigger the execution of the payload.

The basic concept of our integrity analysis is similar to the confidentiality analysis and again based on *AV4: SLAT Remapping*. However, instead of remapping a resource to a secret, we remap code within the VM to a payload we inject from the outside. In detail, our method consists of the following three steps (Figure 4.5):

1. **Trigger Identification.** Identify a code region inside the VM, the execution of which we can trigger from the outside.
2. **Payload Destination Identification.** Send a network packet with a malicious payload to the VM and identify its location inside the VM's memory.
3. **Payload Execution.** Modify the SLAT in order to point the identified trigger to the injected payload and trigger its execution.

To identify our trigger (1), we again make use of *AV1: Page Access Tracking* to perform execute tracking (Section 4.2.1) on all of the VM's GFNs. Next, we exploit a new attack vector, *AV5: Interrupt Injection*, and use the hypervisor's ability to inject a Non-Maskable Interrupt (NMI) into the VM. An NMI is an interrupt that cannot be ignored or disabled by the CPU [67]. This interrupt allows us to determine the GFNs holding the NMI handler and the `do_nmi()` function, the latter of which we intend to use as trigger.

To identify the location of network packets within the VM's memory (2), we make use of our last attack vector, *AV6: Unprotected DMA Channel*. This attack vector exploits the hypervisor's ability to observe the I/O operation when copying the packet from the shared into private memory. Combined with detailed knowledge about virtio, this enables us to determine the exact location of every incoming network packet within the VM's memory.

To conclude our attack, we inject a network packet containing our payload at a carefully calculated offset into the VM. Afterwards, we again exploit *AV4: SLAT Remapping* and abuse the missing integrity protection to modify the SLAT managing the mapping between GFNs and SFNs. Specifically, we map the GFN containing the trigger to the

SFN of the payload (❸). Finally, by injecting an NMI into the VM, we trigger the execution of our payload.

Note that we do not limit our attack to virtio or Linux. Even though our work uses Linux and virtio as concrete examples, the attack vectors upon which we base the attack are completely independent from the OS, subsystem and driver. Thus, targeting another OS or subsystem would not change the essence of the three steps (❶ – ❸), which permits us to breach the VM's integrity.

4.3.1 Trigger Identification

AMD's virtualization extensions allow the system to pass interrupts directly to the VM. Alternatively, the hypervisor can decide to intercept an interrupt and subsequently inject a *virtual interrupt* into the VM through the event injection interface [4]. Just as real interrupts, virtual interrupts cause VMs to pause the currently executing code and handle the incoming event urgently in the associated interrupt handler. Consequently, virtual interrupts present themselves as good candidates that we can abuse for triggering code inside a VM. Hence, they represent our next attack vector: *AV5: Interrupt Injection*. Yet, most interrupts are vital to the VM's execution and temporary replacing the interrupt's code with our payload would crash the VM. Therefore, we prefer to resort to a specific interrupt type which is rarely generated, the NMI. Note that alternatively, we could also resort to injecting other rare interrupts or exceptions, such as the software breakpoint exception that is generated upon executing the `int3` instruction.

At an interrupt, the CPU uses the Interrupt Descriptor Table, a hardware-defined data structure that registers all interrupt vectors, to locate the NMI handler. Once the hypervisor injects a virtual NMI into a VM, the hardware immediately interrupts the VM's execution and invokes the NMI handler. In the context of our attack, we can abuse this behavior to initiate the execution of arbitrary code residing at the location of the registered NMI handler. Specifically, if we remap the GFN of the NMI handler to the SFN containing our payload (Figure 4.5) and issue a virtual NMI, we will be able to execute our payload.

Consequently, in order to perform the remapping, we have to identify the GFN holding the NMI handler (❶). For this, we assume that the hypervisor has access to the exported symbol information or to the unencrypted image of the VM's kernel. This is a valid assumption, as VM images are often provided by the cloud provider and the unencrypted kernel image usually resides in `/boot`. By analyzing the kernel image, we can determine the offset of the NMI handler. We discuss alternatives for determining the offset in Section 4.3.5. However, Linux systems by default make use of KASLR to randomize the offset of the kernel image in the VM's virtual and physical address space [43, 140]. Fortunately, we can make use of different approaches that allow us to bypass KASLR and hence determine the GFN of the NMI handler. Specifically, we have developed the following three methods:

Manipulating KASLR On modern AMD platforms, the Time Stamp Counter counts the number of processor-clock cycles since the last reset [4]. The current value of the counter can be read via the `RDTSC` instruction. This instruction is used by the Linux kernel to create entropy at boot time, which is among others required to initialize KASLR. Using this knowledge, we showed in one of our additional publications that a malicious hypervisor manipulating the results of the `RDTSC` instruction at boot time will be able to pin the KASLR offset [124]. This enables us to guess the KASLR offset used by the VM. Yet, the manipulation of the `RDTSC` instruction requires a malicious hypervisor to actively interfere with the VM at boot time. Additionally, the VM would be able to detect the static KASLR offset. To circumvent these limitations, we can also resort to techniques which allow us to infer the KASLR offset at run-time.

Fingerprinting Nested Page Faults When analyzing the VM's boot processing using *AV1: Page Access Tracking* for another additional publication, we observed that the VM's memory accesses during system boot are deterministic [148]. This observation can assist us in bypassing KASLR. Specifically, in the initial stage of the boot process, before the OS within the VM activates KASLR, the VM exhibits the same GFN access pattern across reboots. Only once the boot process starts to decompress the kernel image to a random offset in the VM's physical memory, the sequence will start to differ. We determined that the first GFN which differs from this sequence holds the beginning of the kernel image in the VM's physical memory. As the beginning of the kernel image is always aligned on a 2 MB boundary [130], we can safely assume that the start of the image will always remain at page offset zero.

This method permits the hypervisor to passively analyze the GFN access pattern of the VM's early boot stage. Yet, since the attacker does not necessarily have the chance to observe the bootstrapping phase of the VM, we have developed another alternative method.

Probing the NMI Handler We can also combine *AV1: Page Access Tracking* with *AV5: Interrupt Injection* to identify the GFN holding the NMI handler. Specifically, performing execute tracking on all of the VM's GFNs will cause the VM to trap into the hypervisor on every instruction fetch. Using this configuration, we use *AV5: Interrupt Injection* and inject a virtual NMI into the VM.

Upon resuming the VM, it will directly jump to the respective NMI handler and immediately trap into the hypervisor, enabling us to locate the GFN of the NMI handler. Shortly after resuming the VM, it will trap again upon calling the `do_nmi()` function responsible for dispatching NMI's. The second trap is guaranteed by the fact that the `do_nmi()` function belongs to a different section than the NMI handler, namely the `.text` section instead of the `.entry.text` section. Therefore, we can be sure that the functions are located on different GFNs and that after trapping for the NMI handler, the VM will trap again for the `do_nmi()` function. In other words, both functions reside on different GFNs and thus guarantee two consecutive traps.

Once we have identified the GFN holding the `do_nmi()` function, all that is left to determine is the exact offset of the function within the respective GFN. This knowledge allows us to inject our payload at the same offset, ensuring it will be immediately executed after we modify the SLAT and trigger its execution. To determine the offset of the `do_nmi()` function, we could refer to the VM's register state available to the hypervisor on `VMEXIT`. Yet, on AMD SEV-ES, the system encrypts the VM's register state on every `VMEXIT` and hence obfuscates the offset. Thus, we instead determine the offset of the `do_nmi()` function by analyzing the exported symbol information of the VM's kernel image.

4.3.2 Payload Destination Identification

The architectural constraints of SEV prohibit devices to apply Direct Memory Access (DMA) into encrypted VM memory (Section 2.1). This limitation applies to physical as well as emulated devices, forcing VMs and devices to exchange data through shared pages. Even though this security-driven design decision aims to protect the system against DMA attacks [11, 16], it also establishes a vantage point, our *AV6: Unprotected DMA Channel*. To be precise, the unprotected DMA channel facilitates attackers to extract or inject arbitrary data from or to the VM, or to inject arbitrary code into the VM's memory without knowing the secret encryption key. While previous attacks have targeted the former [91], we focus on the latter attack vector. Specifically, we abuse the exposed network communication channel to inject arbitrary code into the encrypted memory.

Yet, before injecting the payload, we need to identify where in memory the VM stores incoming network packets (❷). For this, we record the sequence of GFNs the VM is writing to when bouncing an incoming network packet from shared into private memory (Section 2.1). Specifically, after the device in the hypervisor places a network packet into a buffer in shared memory, we perform access tracking on the GFN holding the buffer. Hence, the hypervisor gets notified as soon as the VM tries to access the buffer. Having received this notification, we conclude that the VM is about to read the buffer. Next, we enable write tracking on all other GFNs. This allows us to collect the sequence of GFNs the VM is writing to until we can be sure that it has finished bouncing the packet into the private memory. In order to determine when the VM has finished bouncing the packet, we abuse the notification system of virtio.

Using virtio, the hypervisor raises an interrupt to notify the driver that it can consume a buffer. Yet, if the driver is busy processing other data, the virtio specification grants the driver in the VM the possibility to inform the hypervisor that it does not want to receive interrupts. In older virtio implementations, the driver informs the device by setting the `VRING_AVAIL_F_NO_INTERRUPT` flag. By unsetting the flag, the driver indicates that it has finished processing the data. In recent virtio implementations, the driver uses a special index of the available ring, the `used_event`, to indicate the last consumed buffer. Once the driver has finished consuming a buffer, it updates the `used_event` to the index of the consumed buffer. By monitoring both the `VRING_AVAIL_F_NO_INTERRUPT`

flag and the `used_event` entry, we comply with all virtio versions. This permits us to determine when the driver has completed processing the packet.

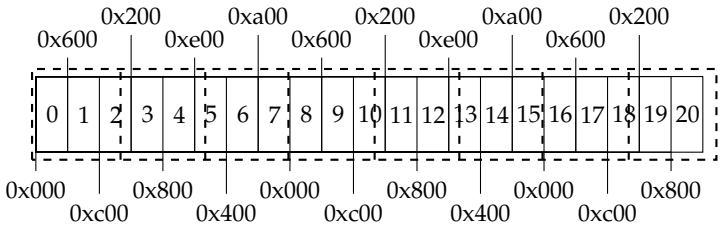

Figure 4.6: The 32 KB sized packet buffer contains 21 fragments of `0x600` bytes. The size of 32 KB requires eight GFNs, indicated by the dashed lines. Source: [109].

Having determined the write sequence when bouncing a network packet into the VM, we next identify which GFN in the sequence contains the packet and at which offset the packet is located. To determine the exact position of a packet, we have to understand how virtio manages its buffers. The buffers are allocated in 32 KB chunks of contiguous memory called a *packet buffer*. Each packet buffer is divided into multiple fragments of the same size. Figure 4.6 shows the layout of a packet buffer holding 21 fragments of `0x600` bytes each. The fragments are aligned and padded to `0x600` bytes, with the first fragment starting at offset `0x0`. The dashed lines indicate the eight GFNs holding the buffer.

It is important to note that within one packet buffer, all fragments have the same size. Yet, the size of the fragments can be dynamically adjusted to consider the size of recent network packets [36]. Therefore, the number of fragments per packet buffer can vary. During our test, we always observed a fragment size of `0x600` bytes. This size can change when the VM starts to receive larger frames. We can detect changes to the size of the fragments in the packet buffer by monitoring the length of the buffers in the virtqueue's descriptor table.

This knowledge of the packet buffer's structure allows us to infer which GFNs contain which fragments, and which fragment starts at which offset. First of all, we know that the packet buffer is 32 KB aligned in the VM's memory. Hence, we search the write sequences performed by the VM when bouncing a packet into private memory for a GFN which is 32 KB aligned. This GFN represents a candidate for the first GFN of a newly allocated packet buffer. Knowing that the first GFN contains the start of three fragments (Figure 4.6), we confirm the candidate by analyzing if the VM also writes to the GFN when bouncing the following two packets into private memory.

We can repeat this process for the subsequent GFNs in the packet buffer to minimize false positives. For this, we make again use of our detailed knowledge of the packet buffer. For example, the second GFN of the packet buffer contains the start of fragments 3, 4 and 5. Therefore, assuming we have correctly identified the first GFN of the packet buffer, the second, subsequent GFN should be accessed in the fourth, fifth, and sixth

bouncing sequence. The third GFN containing the start of fragments 6 and 7 should be accessed in the sixth and seventh bouncing sequence, and so on.

Note that depending on the size of the incoming network packet, the VM may also access the second GFN in the third sequence. Considering the minimum size of `0x40` bytes for an Ethernet frame [65], the packet stored in the third fragment will not cause a write access to the second GFN. Yet, a packet larger than `0x400` bytes will cause the VM to also write data to the second GFN, as the fragment offset of `0xc00` plus the packet size of `0x400` cross the boundary of the first GFN.

By applying these conditions and adapting them in order to reduce the number of false positives, we are able to successfully identify the packet buffer within the VM's memory. Our tests show that requiring the first ten write sequences of the bounce buffer to fulfill our conditions eliminates the chance for false positives on all tested VMs. Having determined the position of the packet buffer, we can also determine the offset of each fragment by knowing its size. This allows us to predict where the VM will store prospective incoming network packets.

Also note that the VM copies incoming network packets into the packet buffer before it applies any processing or filtering. Thus, the VM also stores packets sent to closed or even firewalled ports in the packet buffer. This enables us to send our payload to closed or filtered ports and, in comparison to our attack targeting the VM's confidentiality (Section 4.2), eliminate the need for any active network service.

4.3.3 Payload Execution

Having determined the trigger point and the location to which we aim to inject our payload, we finalize the setup and conclude our attack by executing an injected payload (❸). This final step of our attack comprises two stages. First, we inject a payload into a suitable fragment in the packet buffer. Second, we manipulate the SLAT to remap the trigger such that its GFN translates to the SFN holding the injected payload, again exploiting *AV4: SLAT Remapping*. Finally, after modifying the SLAT, we use *AV5: Interrupt Injection* and inject an NMI to execute the payload.

When injecting the payload into a suitable fragment, we have to consider that the trigger and the fragment will likely reside on different page offsets. As such, we must place the payload into a fragment whose location in memory spans a range that comprises the needed page offset. In the example depicted in Figure 4.7, the trigger is located at the page offset `0x700`. Thus, we choose to inject the payload to a fragment that starts at offset `0x600`. To position the payload at precisely the same offset as the trigger, we fill the first `0x100` bytes of the network packet we send to the VM with arbitrary data. Since the fragment also holds protocol headers of the packet, the arbitrary data only spans the size of `0x100` bytes minus the size of the protocol headers, such as Ethernet frame, IP and, TCP or UDP header. After a total of exactly `0x100` bytes, we place our payload to ensure that the payload and the trigger are located at the same page offset.

For the next step, we again make use of the fact that while SEV's memory encryption protects the guest page tables within the VM, the hypervisor remains in charge of the

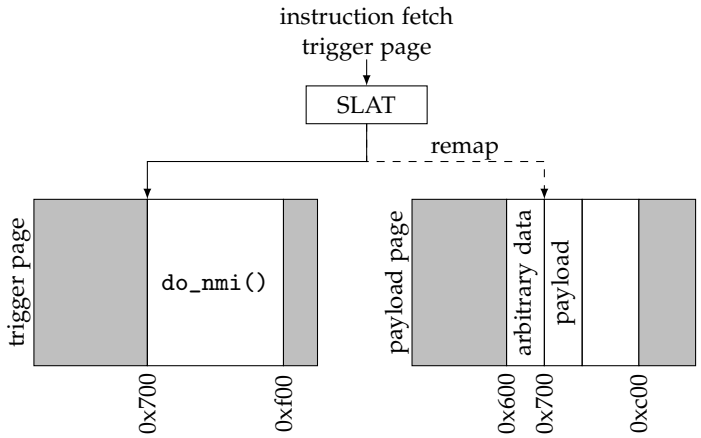

Figure 4.7: The system maps the GFN of the page of the trigger to the original SFN holding `do_nmi()`. We leverage the SLAT to remap the GFN of the trigger page to another SFN holding the injected payload. Thus, by injecting an NMI, we manage to redirect the control flow to our payload. Source: [109].

SLAT. Specifically, once we prepared and injected the payload, we adjust the respective entry in the SLAT such that the GFN that initially translated to the SFN holding the trigger translates to the SFN holding the injected payload. That is, once the NMI handler attempts to call the `do_nmi()` function, it will instead execute our payload.

4.3.4 Evaluation

To show that our attack breaching the VM's integrity is effective and OS-agnostic, we created a proof of concept and evaluated its precision on different VMs.

Setup

Our setup comprised a Debian 11 host running a custom-built Linux kernel `v5.6` including the unofficial patches for SEV-ES[5]. The host equipped an 8-core EPYC 3251 CPU with 64GB of RAM and used QEMU `v4.2.50` and KVM to virtualize the targeted VMs. Furthermore, we extended the host's kernel with our SEVered framework[6] (Section 4.2.5). Additionally to the functionality already required for our attack on the VM's confidentiality (Section 4.2), we added code for the exploitation of *AV5: Interrupt Injection* and *AV6: Unprotected DMA Channel* to the framework. Specifically, the framework allows to inject NMIs into the VM. Hence, we can determine the address of the VM's interrupt handler by performing execute tracking on all pages and afterwards injecting an NMI (Section 4.3.1). Furthermore, we included functionality that exploits the unprotected

[5] `https://github.com/AMDESE/AMDSEV`
[6] `https://github.com/Fraunhofer-AISEC/severed-framework`

DMA channel to identify the packet buffer via a combination of memory access and write tracking.

Having added this functionality to our SEVered framework, we evaluated our attack by targeting SEV-protected VMs running the following five different Linux distributions officially supported by AMD [5]: SUSE Linux Enterprise Server (SLES) 15, RedHat Enterprise Linux (RHEL) 8, Fedora 29, Ubuntu 18.04, and openSUSE-Tumbleweed. To enable SEV-ES, all VMs were running a custom built kernel in version `v5.7.0` with patches for SEV-ES support[5].

Precision Evaluation

To evaluate the precision of our attack, we ran it 1,000 times against all five VMs. Before every round of the attack, we sent a random number between 0 and 64 packets with random data to avoid any bias from previous rounds. Afterwards, in every round of the attack, we identified the GFN of `do_nmi()` by injecting an NMI and performing execute tracking (Section 4.3.1). To identify the packet buffer in the VM's memory, we sent dummy packets and analyzed the page fault sequence for bouncing the packets into the VM's private memory. Once a series of ten page fault sequences fulfilled our conditions, we deemed the packet buffer identified. While a smaller number may have increased the performance of the evaluation, the higher number allowed us to eliminate false positives. Having identified the packet buffer, we sent additional dummy packets in case the next free fragment was not located on a suitable offset to inject our payload (Section 4.3.2). The payload itself executed a hypercall with exit reason `0xff`, and afterwards issued a `RET` instruction in order to return to the NMI handler. Having performed all required preparations, we remapped the GFN of `do_nmi()` to the SFN of the payload and triggered its execution. As hypercalls immediately trap into the hypervisor, we were able to determine the successful execution of the payload by monitoring for hypercalls with exit reason `0xff`. To conclude the round, we connected to the VM via SSH to ensure that it continued to behave as expected.

To perform the evaluation itself, we filtered any external traffic to the VM in order to maintain full control over incoming network packets. The main reason for this approach is the fact that our current implementation analyzes the VM's page fault sequence after the VM has bounced the network packet into its private memory. By that time, the VM might already have received additional packets. The filtering of traffic permits us to ensure that the VM received the next packet only after we finished analyzing the page fault sequence. To deal with simultaneously arriving packets and to further improve the stealthiness of our attack, we can adjust our proof of concept to buffer incoming packets until we complete the analysis of the page fault sequence.

Note that in a real-world setting, incoming packets would have allowed us to determine the exact destination of the payload in a stealthy way, without having to inject any artificial packets. Yet, to avoid having to rely on fluctuating numbers of incoming network packets, which would have influenced the evaluation results, we have opted for completely controlling the network traffic.

Table 4.5: Evaluation details of 1,000 runs of our attack against different VMs protected by SEV. The table further summarizes an averaged number of the tracked write accesses to the VM's memory when bouncing network packets, the number of the sent packets, and the duration of the attack. Source: [109].

VM Image	Number of Write Accesses	Number of Sent Packets	Attack Duration [s]	Success Rate
SLES	38.52	19.97	2.57	100 %
RHEL	66.78	19.50	2.75	100 %
Fedora	68.74	20.37	3.04	100 %
Ubuntu	43.10	19.89	2.65	100 %
openSUSE	40.57	19.56	2.57	100 %

For each VM, we conducted 1,000 iterations of our attack, achieving a total success rate of 100 %. Table 4.5 shows the detailed results of our evaluation. In the second column, we show the average number of write accesses recorded when monitoring the VMs bouncing a packet from shared to private memory. Their values range from 38.52 for openSUSE to 68.74 for Fedora.

In the third column, we show the average number of packets we had to send before we could identify the packet buffer in the VM's private memory. To ensure that we always correctly identify the packet buffer, we required 10 consecutive write sequences to match our conditions. In our test environments, all VMs filled the packet buffer with 21 fragments of `0x600` bytes. Therefore, on average, we had to send 10 packets before a new packet buffer was allocated. The results in the third column match the expected value of these 10 packets plus 10 packets to ensure the packet buffer identification was correct.

The fourth column shows the average time needed to identify the packet buffer (❷), inject our payload, remap and trigger its execution (❸). The values are almost all below three seconds, ranging from 2.57 seconds on SLES and openSUSE to 3.04 seconds on Fedora. From these results, we conclude that while filtering external traffic during the attack may be noticeable by the VM, the duration is not sufficient to attract specific attention. Additionally, extending our implementation to buffer incoming packets for a short period of time instead of filtering external traffic would further increase stealthiness, as we would not need to send artificial packets. This leaves the VM with the possibility to analyze page access times to determine if the hypervisor is performing memory access tracking. However, as we only require a maximum of around three seconds for our attack, it would be difficult to distinguish between delays caused by access tracking and other activities, such as migration of the VM.

The last and most important column shows our success rate of 100 % on every single VM we run our attack against. This confirms the identification of the trigger page

(Section 4.3.1) and the payload (Section 4.3.2) work as described, enabling us to execute our attack against different SEV-protected VMs with a success rate of 100 %.

4.3.5 Discussion

Having analyzed the precision of our attack against the VM's integrity, we discuss details of our attack as well as possible further improvements.

Kernel Image Availability We assume that the hypervisor has access to the kernel image to extract the kernel's symbol information (Section 4.3.1). This is a valid assumption, as the kernel image typically resides in the unencrypted `/boot` partition of the VM's virtual disk image or, in some situations, can be provided by the hypervisor. In both cases, the hypervisor gains direct access to the kernel image. In case the VM stores its kernel image in an encrypted `/boot` partition, an attacker can make use of other methods to gain access to the VM's kernel image. For example, SEV supports and disseminates only a small set of Linux distributions, each running a specific kernel version that is known to the public [5]. In other words, assuming the VM's provider did not compile a custom Linux kernel, the attacker can download and inspect the provided Linux kernel images to collect the necessary information to identify the trigger point. Alternatively, we could apply OS fingerprinting mechanisms [20, 147] to determine the used kernel version.

Bypassing KASLR We showed one approach to practically disable KASLR, and two approaches to identify the location of the `do_nmi()` function within the VM's memory with KASLR enabled (Section 4.3.1). Both identification approaches work with the most recent implementation of KASLR. Yet, the granularity of KASLR might change in the near future. In a recent patch, Accardi [42] proposed to also randomize the offsets of functions within the kernel image. This would not require any changes to our trigger point identification (Section 4.3.1) with SEV, as the VM executes its NMI handler directly after an interrupt is injected. In comparison, using SEV-ES, after injection of an interrupt, the VM first internally handles the interrupt. This may cause the VM to access different GFNs before handing control to the NMI handler. To still be able to determine the location of the NMI handler and the `do_nmi()` page, we could analyze the page faults caused by the VM's internal handler to determine when it finishes execution and hands control to the NMI handler. Alternatively, we could also resort to machine learning [20] to determine the location of the `do_nmi()` function.

Injected Code Size Our prototype injects a small payload which executes a hypercall that allows us to determine the success of the attack. Furthermore, we also experimented with payloads up to 78 bytes. Yet, to inflict significant damage, the attacker may prefer to execute arbitrary code. However, the Maximum Transmission Unit defines the packet size and presents an obstacle to our approach. Yet, by injecting multiple payload fragments and using `RIP`-relative jumps, we can transfer the control flow to other payload segments

located in different fragments. This way, we can inject and execute payloads of a nearly arbitrary size.

Stealthiness Our prototype is not designed to be stealthy. Yet, with some modifications, we can reduce its visibility. As part of our attack, we remap the GFN that holds the `do_nmi()` function to the SFN holding our injected payload. Since we can cause the VM to execute the `do_nmi()` function from outside of the VM, it acts as a trigger for executing the injected payload. To still ensure consistent VM operation after remapping, we continue tracking the execution of incoming NMI requests by monitoring execute accesses to the GFN containing the NMI handler. That is, every time a regular NMI request occurs, we can restore the original mapping in the SLAT and continue execution. Yet, walking the SLAT tables to change permissions of GFNs on every NMI event is a slow operation and can lead to race conditions in multi-virtual CPU (vCPU) environments. Besides, changes in the VM's memory are visible to other vCPUs. To accommodate these issues, we can prepare a set of SLAT tables which define different mappings on the VM's physical memory, and rapidly switch among them without revealing the in-memory artifacts [151, 122, 123]. Since we can individually assign a SLAT table to each vCPU, the remaining vCPUs would not be able to detect the adjustments in memory.

Another possibility to increase stealthiness is the current approach of sending UDP packets to a closed port. In particular, this could allow the VM to detect the unusual amount of suspicious traffic to a closed port and reveal the attack. To further increase the stealthiness of our prototype, we could fall back to sending less suspicious packets, such as ARP packets. While ARP packets are limited in size, we can use them to detect the packet buffer and to fill unsuited packet buffer fragments with dummy packets.

Additional Network Traffic Even though our prototype blocks external traffic to the VM (Section 4.3.4), we underline that this strategy is not a requirement. On the contrary, additional network traffic would facilitate detecting the packet buffer in the encrypted memory. This way, we would not need to add any artificial traffic to observe the sequences of write accesses of the bounce buffers. The same applies to identifying a suitable fragment in the packet buffer. For one, it would eliminate the need to send dummy packets to observe sequences of write accesses during the bounce processes. Furthermore, we could avoid having to send packets in case the next free fragment is not located at a suited offset. Hence, analyzing external traffic instead of a stream of artificial packets increases the stealthiness of the attack. In this scenario, the attacker would have to send only a single packet with the payload. To avoid issues caused by multiple network packets received shortly after each other, an improved implementation of our proof of concept can queue packets until analysis steps such as access page tracking of previous packets are finished. Hence, our attack can also be applied to VMs with high network load. However, we opted for an evaluation which did not require artificial network traffic in our test environment as this could have influenced our evaluation results.

Generalization While we evaluated our attack by injecting a payload via network packets, we neither rely on the usage of virtio or Linux, nor the network communication channel. Even though we use Linux and virtio as concrete examples, the actual vulnerability is entirely independent from the OS, subsystem and driver. Assuming that another OS, such as Windows, would provide support for SEV, the system would still be architecturally forced to transfer data from DMA-capable devices into the protected VM's memory. The system will achieve this transfer by copying the data from unencrypted to encrypted memory, which we exploit to inject our payload. Therefore, targeting another OS, subsystem or driver would not change the fundamentals of the three steps of our attack.

Yet, SEV is currently only supported by Linux, and most Linux-based virtualization environments use virtio. For both Linux and virtio, our prototype does not require a specific version. To be precise, by monitoring virtio's `VRING_AVAIL_F_NO_INTERRUPT` flag as well as its `used_event` index, it is compatible to older as well as newer versions of virtio. Additionally, to confirm that we do not impose any requirements on the version of the VM's kernel, we tested our attack on different kernel versions between `v4.19.43` and `v5.7.0`. The kernel version is only limited by having to support SEV, which has been integrated into the Linux mainline kernel `v4.16` [85].

4.4 Summary

In this chapter, we investigated our first research question, *RQ1: Which attack vectors enable attacks against a TEE's cryptographic memory isolation without memory integrity?*. For this investigation, we presented three contributions and established total of six attack vectors. The first two contributions are attacks showcasing the impact of missing memory integrity on the example of AMD SEV. Our attacks confirm that cryptographic memory isolation cannot hold without memory integrity, preventing TEEs from protecting themselves from a curious or malicious administrator. To be precise, the first attack exploits the missing integrity protection to extract confidential data from a VM protected by SEV. Similarly, the second attack uses the same weakness to execute arbitrary code within the VM. In a detailed evaluation, we have shown that both attacks work reliably.

The only major requirement for our contribution *C1*, a data extraction attack, is the presence of a service within the VM providing a resource to the outside. As VMs are typically used in cloud environments, they are required to provide such a service to be able to communicate with the outside world. Our attack uses this service to repeatedly request one of its resources. By combining this step with *AV1: Page Access Tracking*, we can locate the resource within the VM's memory. Having identified the resource, we use *AV2: Network Monitoring* and *AV3: Disk Access Monitoring* to track the VM's page accesses until we detect an event indicating that the VM recently accessed a secret. This step provides us with a list of pages which are likely to contain the secret. Next, we make use of *AV4: SLAT Remapping* in order to systematically retrieve pages from this list. By simultaneously analyzing the content of the retrieved pages, we are able to quickly

extract the respective secret from the VM's memory despite it being protected by SEV. To demonstrate the feasibility of our approach, we evaluated our prototype with different services, namely Apache, nginx and OpenSSH. For every service, we tested various levels of concurrent accesses to evaluate our attack under different realistic noise conditions. Our results show that we are able to extract confidential information such as private keys and FDE keys from the protected VM within seconds with a very high success rate. Additionally, as our attack is independent of the specific service, our method can easily be adapted to a variety of different attack scenarios. As such, we demonstrate that a malicious hypervisor still remains able to extract sensitive data from SEV-protected VMs.

Our contribution *C2* is an attack that allows us to inject and execute truly arbitrary code into SEV-protected VMs. Also this attack builds upon SEV's missing memory integrity protection and the resulting *AV4: SLAT Remapping*. Additionally, it makes use of *AV6: Unprotected DMA Channel*, caused by SEV's inability to perform DMA into encrypted memory. In comparison to our first attack, our code injection attack does not require any services running in the VM. Instead, we abuse the fact that Ethernet frames are pulled into the VM's memory before any filtering mechanisms can apply. Also, it suffices if our packets are copied into the VM's memory only for a short period of time. To perform the attack, we first identify the location of code within the VM which we can trigger from the outside using *AV5: Interrupt Injection*. Next, we inject a network packet into the VM containing an arbitrary payload and use page tracking to identify the location of the payload within the VM's memory. We then remap the GFN of the trigger to the SFN of the page containing the payload. Finally, we provoke the execution of the trigger, which will cause the VM to instead execute our payload due to the modified memory mapping. To evaluate the effectiveness of our approach, we run our attack against various SEV-protected VMs. Our evaluation demonstrates a success rate of 100 %, while at the same time only requiring a few seconds. As the data extraction attack, also this attack is independent from a specific service, OS and subsystem. Hence, our attack shows that additionally to being able to breach the VM's confidentiality, a malicious hypervisor can also interfere with the integrity of an SEV-protected VM.

To determine if upcoming VM-based TEEs correctly verify the integrity of their memory, our contribution *C3* is a framework supporting the different analysis methods we used for our attacks. For example, the framework permits to track a VM's memory accesses or to modify its memory mapping.

Overall, our example attacks highlight that TEEs require to protect the integrity of their memory to be able to cryptographically isolate their memory from a curious or malicious administrator. Hence, the current version of SEV also cannot protect against an adversary gaining administrative access by escaping an isolation domain [143, 136, 30, 31] hosted on the same physical system. In other words, the protection offered by SEV in its current state is no stronger than those of traditional VMs, as an administrator will still be able to read or modify the VM's data. However, this does not imply that SEV is no security improvement. Instead, different countermeasures could protect against

the presented attacks, allowing SEV to successfully protect the VM's confidentiality and integrity. In the following chapter, we analyze such countermeasures against our attack vectors.

CHAPTER 5

Restoring Cryptographic Memory Isolation without Memory Integrity

In the previous chapter, we discussed two example attacks that violate AMD Secure Encrypted Virtualization (SEV)'s cryptographic memory isolation. Having determined the six different attack vectors the attacks rely on, we now continue by discussing how the cryptographic memory isolation could be restored by remedying one or more of these vectors. For this investigation, we stick to the example of AMD SEV and analyze different software and hardware countermeasures against our attacks that can be applied by both the Virtual Machine (VM) and SEV. Specifically, we first discuss countermeasures that can be applied by the VM itself without requiring an update of SEV. These countermeasures are mainly workarounds which do not resolve the most critical attack vectors, but aim to make the attack more difficult. To effectively resolve these critical attack vectors, an update to SEV is inevitable. We therefore also discuss how our attacks could be prevented by modifying SEV. Most importantly, we present our contribution *C4*, a countermeasure against *AV4: SLAT Remapping*. While we already mentioned the idea for this countermeasures in one of our publications [107], we will use this chapter to discuss in detail how it could be implemented. For our contribution, we propose to modify the tweak mechanism applied by SEV's memory encryption to also include the Guest Physical Address (GPA). This requires only a small modification of SEV, while at the same time preventing both our attacks as well as others exploiting this attack vector [60]. As the modification of the tweak mechanism would have to be implemented by AMD, we only discuss the countermeasure on a theoretical level.

5.1 Countermeasures by the VM

The first requirement for both attacks is the possibility to perform different variations of page tracking, *AV1: Page Access Tracking* (Section 4.2.1). Therefore, preventing a malicious hypervisor from tracking the VM's page accesses would prevent both attacks. In particular, the hypervisor would not be able to identify the Guest Frame Numbers (GFNs) containing relevant data such as the resource, secret, trigger or payload. In

order to prevent the hypervisor from performing page tracking, it would be necessary to revoke its ability to modify Page Table Entrys (PTEs) in the Second Level Address Translation (SLAT). However, the hypervisor requires this ability to perform memory management, for example to indicate that a page has been swapped out of main memory. To the best of our knowledge, the only possibility to address this issue would be to either split the hypervisor into a trusted and an untrusted part [154, 92] or to introduce a separate component responsible for memory management [150] (Section 3.2). Yet, this does not address the attack vector itself, but only moves the trust away from the hypervisor to a different component.

Instead, the VM could try to reduce the information leaked by its memory access pattern. This could be achieved using Oblivious RAM (ORAM), which randomizes data access patterns [55, 84, 137]. Such a randomization would make it harder for an attacker to identify the page containing the different GFNs. However, a downside of ORAM is that it requires constant shuffling of the data, causing a high overhead. Similarly, the VM could try to reduce the information leakage from page faults by deploying deterministic multiplexing [133]. Using this approach, programs show a deterministic memory access behavior independent from the input. Applying either ORAM or deterministic multiplexing to our scenario would complicate the identification of valuable information within the VM's memory. However, the VM would nevertheless be required to access the data, which can be observed by the hypervisor. Therefore, both approaches would not be able to hide the location of valuable information, but only complicate the identification of the respective GFNs.

An alternative approach would be to reduce the impact of *AV1: Page Access Tracking* by purging critical information as soon as it is no longer required to prevent its extraction from the VM's memory. In particular, during our tests we discovered that the private keys of the web servers as well as the Full Disk Encryption (FDE) keys remained at the same memory location. In comparison, the SSH daemon forked a new process for every incoming connection. This fork causes the private key used to establish the connection to also be located in the memory of the new process. Therefore, as soon as the connection is closed and the process is terminated, the private key can be purged from memory. In other words, the extraction of the private key is only possible as long as the SSH connection remains open, thereby reducing the time frame for a possible attack. However, our evaluation shows that we require less than 1.5 seconds to extract the private key from the SSH server. Hence, it is not sufficient to purge the key once the connection is closed. Instead, the process could overwrite the key as soon as it is no longer required, which should be the case after successfully completing the SSH handshake. A similar approach could be taken by the web servers. Yet, for some secrets, such as FDE keys, purging might not be feasible. To be precise, as the VM requires the key for every disk I/O operation, it has to remain accessible.

For secrets which need to be accessible permanently, one may consider storing them outside of the VM's memory in dedicated hardware. Specifically, as our first attack is only able to extract secrets from memory, secrets stored for example in a Hardware

Security Module would be protected from such extraction. Similarly, storing FDE keys in memory can be avoided by deploying hardware based disk encryption. However, this countermeasure does not defend against the attack itself, but merely reduces its impact, as the adversary will not be able to extract the respective secrets from memory. Hence, this approach should only be seen as a workaround.

Addressing *AV2: Network Monitoring*, *AV3: Disk Access Monitoring* and *AV5: Interrupt Injection* faces similar limitations as the prevention of *AV1: Page Access Tracking*. Specifically, the hypervisor requires access to the VM's network traffic in order to forward packets to the VM and to the outside world. Similarly, as the hypervisor is also responsible for providing a virtual disk to the VM, it is difficult to prevent it from monitoring the VM's disk accesses. Furthermore, removing the hypervisor's ability to handle interrupts would disable an essential virtualization component. Hence, the respective attack vectors could only be rectified by moving their underlying tasks, which are network, disk and interrupt management, to other, trusted entities. Yet again, this approach would not address the attack vector itself, but only move trust to a different component.

Instead of interfering with the foundations of virtualization, countermeasures should focus on attack vectors specific to SEV, namely *AV4: SLAT Remapping* and *AV6: Unprotected DMA Channel*. To prevent *AV4: SLAT Remapping*, the VM could attempt to implement its own integrity protection. For this, it could store integrity measurements of data in its memory, and verify the respective measurement before using the data. To ensure the integrity of the measurement itself, it could be stored in a secure location outside of the memory, such as a Hardware Security Module. However, such an approach faces multiple challenges. For once, it requires high amounts of storage to store all integrity measurements. Additionally, verifying the measurement in software before each memory access will likely have a high performance impact. Hence, we think that changes in the design of SEV Encrypted State (SEV-ES) are required in order to effectively prevent *AV4: SLAT Remapping*, which we discuss in the next section.

Finally, to prevent *AV6: Unprotected DMA Channel*, the VM could avoid copying data from shared to private memory. It could achieve this by performing read operations on the shared memory, while performing write accesses on private memory. In the example of an incoming network packet, the VM could decrypt TLS-content to its private memory, but consume other contents of the packet directly from shared memory. This approach would make it difficult to determine where in its private memory the VM stores the decrypted content of the network packet. However, such an approach would require to modify existing software in order to perform read operations on shared memory, but write operations to private memory.

5.2 Countermeasures by SEV

The discussion of possible countermeasures that could be performed by the VM indicates that complete protection from our attacks can only be achieved by modifying the design of SEV. Specifically, both our attacks rely on *AV4: SLAT Remapping* and break SEV's

cryptographic memory isolation by remapping the VM's GFNs to different System Frame Numbers (SFNs). To be precise, our attack against the VM's confidentiality remaps the GFN of the resource to the SFN of the secret (Section 4.2). Similarly, to attack the VM's integrity, we map the GFN of the trigger to the SFN of the payload (Section 4.3).

To prevent this attack vector, we propose to modify the tweaking mechanism applied by the Secure Processor (SP) before encrypting data [75]. While we already previously mentioned the basic idea for this countermeasure [107], we will use the rest of this section to describe a possible implementation in more detail. Specifically, early versions of CPUs supporting SEV hold a list of static 128-bit tweak values which are applied to the plaintext data [41]. Similarly, we discovered for one of our additional publications that more recent CPUs randomly generate these tweak values on every boot [148]. In both cases, the goal of this tweak values is to prevent an attacker from moving blocks in the system's physical memory [75].

To achieve this goal, the SP XORs the tweak values $t_j, t_j + 1, ..., t_n$ to the original message m depending on the address a at which the data should be stored. Now, let a consist of the address bits $a_{n-1}, a_{n-2}, ...a_0$. Then, the SP applies the following tweak:

$$T(a) := \bigoplus_{j \in \{i \in \mathbb{Z}_n : i \geq 4 \wedge a_i = 1\}} t_j$$

Where t_j is the constant tweak value for the respective bit that is either static or generated at boot. That is, for every bit in the physical address starting from the fourth bit, the tweak XORs the message with the respective tweak value if the bit is set to 1. Note that bits 0 – 3 of the address are ignored, as the 128 bit AES-blocks used by the SP cause plaintext data to be stored in the same ciphertext block independent from these first four address bits.

The SP then uses this tweak to encrypt data in memory based on the System Physical Address (SPA) on which the data will be stored. In detail, we were able to reverse engineer the following encryption mechanism for a message m to be stored in memory [148]:

$$\mathrm{Enc}_K(m, SPA) := \mathrm{AES}_K(m \oplus T(SPA)) \oplus T(SPA)$$

Accordingly, when a ciphertext c is read from memory, it is decrypted as follows:

$$\mathrm{Dec}_K(c, SPA) := \mathrm{AES}_K^{-1}(c \oplus T(SPA)) \oplus T(SPA)$$

This approach prevents an attacker from creating predictable plaintext by moving blocks in the system's physical memory. To be precise, as this would change the SPA of the encrypted data, the SP would apply a different tweak to the decryption, causing it to return garbled data. However, as we have shown in the previous chapter, the tweak does not prevent a malicious hypervisor from remapping GFNs to different SFNs. Specifically, the remapping technique circumvents the protection as it only modifies the GPA, but

not the SPA of the encrypted data. Therefore, we propose to also include the GPA in the tweaking mechanism. This could for example be achieved by repeating the tweaking step with the GPA as input:

$$\begin{aligned}\mathrm{Enc}_K(m, SPA, GPA) &:= \mathrm{AES}_K(m \oplus T(SPA) \oplus T(GPA)) \oplus T(SPA) \oplus T(GPA)\\ \mathrm{Dec}_K(c, SPA, GPA) &:= \mathrm{AES}_K^{-1}(c \oplus T(SPA) \oplus T(GPA)) \oplus T(SPA) \oplus T(GPA)\end{aligned}$$

For simplification, the SP could also first XOR the GPA and the SPA, and then use the result as input for the tweaking function.:

$$\begin{aligned}\mathrm{Enc}_K(m, SPA, GPA) &:= \mathrm{AES}_K(m \oplus T(SPA \oplus GPA)) \oplus T(SPA \oplus GPA)\\ \mathrm{Dec}_K(c, SPA, GPA) &:= \mathrm{AES}_K^{-1}(c \oplus T(SPA \oplus GPA)) \oplus T(SPA \oplus GPA)\end{aligned}$$

This small adjustment allows to incorporate the GPA which the VM aims to access into the encryption. Considering the example of our attack against the VM's integrity, the SP would encrypt the plaintext data m_t holding the code for the trigger as follows:

$$\begin{aligned}c_t =& \mathrm{Enc}_K(m_t, SPA_t, GPA_t)\\ =& \mathrm{AES}_K(m_t \oplus T(SPA_t \oplus GPA_t)) \oplus T(SPA_t \oplus GPA_t)\end{aligned}$$

Where SPA_t is the system physical address of the trigger and GPA_t the guest physical address. During decryption, the different XOR operations with the tweak function neutralize each other, yielding the plaintext:

$$\begin{aligned}m_t =& \mathrm{Dec}_K(c_t, SPA_t, GPA_t)\\ =& \mathrm{AES}_K^{-1}(c_t \oplus T(SPA_t \oplus GPA_t)) \oplus T(SPA_t \oplus GPA_t)\\ =& \mathrm{AES}_K^{-1}(\mathrm{AES}_K(m_t \oplus T(SPA_t \oplus GPA_t)) \oplus T(SPA_t \oplus GPA_t)\\ & \oplus T(SPA_t \oplus GPA_t)) \oplus T(SPA_t \oplus GPA_t)\\ =& \mathrm{AES}_K^{-1}(\mathrm{AES}_K(m_t \oplus T(SPA_t \oplus GPA_t)) \oplus T(SPA_t \oplus GPA_t)\\ =& m_t\end{aligned}$$

Similarly to the code on the trigger page, the SP would tweak the encryption of the code m_p on the payload page with the respective GPA_p and SPA_p:

$$\begin{aligned}c_p =& \mathrm{Enc}_K(m_p, SPA_p, GPA_p)\\ =& \mathrm{AES}_K(m_p \oplus T(SPA_p \oplus GPA_p)) \oplus T(SPA_p \oplus GPA_p)\end{aligned}$$

When the adversary now attempts to change the SLAT in order for the trigger page to point to the payload page, the SP would still tweak the encrypted content with SPA_p for decryption. Yet, it would also include GPA_t into the tweak instead of GPA_p, as the VM intends to access the trigger page.

$$\begin{aligned} m_t =& \mathrm{Dec}_K(c_p, SPA_p, GPA_t) \\ =& \mathrm{AES}_K^{-1}(c_p \oplus T(SPA_p \oplus GPA_t)) \oplus T(SPA_p \oplus GPA_t) \\ =& \mathrm{AES}_K^{-1}((\mathrm{AES}_K(m_p \oplus T(SPA_p \oplus GPA_p)) \oplus T(SPA_p \oplus GPA_p)) \\ & \oplus T(SPA_p \oplus GPA_t)) \oplus T(SPA_p \oplus GPA_t) \end{aligned}$$

As $T(SPA_p \oplus GPA_t))$ and $T(SPA_p \oplus GPA_t)$ do not neutralize each other, the attempt to decrypt c_p would cause the SP to perform the AES decryption on the ciphertext

$$(\mathrm{AES}_K(m_p \oplus T(SPA_p \oplus GPA_p)) \oplus T(SPA_p \oplus GPA_p)) \oplus T(SPA_p \oplus GPA_t)$$

and afterwards XOR the result with

$$T(SPA_p \oplus GPA_t)$$

This creates two challenges for the attacker. For once, without the knowledge of the tweak values used in the tweak function, it is not possible to determine the result of the XOR operations. Additionally, even if the results of the XOR operations would be known, the decryption would nevertheless operate on garbled ciphertext, causing the decryption to also return garbled data. To determine the outcome of the decryption, the attacker would require access to the AES key used for the decryption, which is managed by and only accessible to the SP. Hence, the attacker would have to compromise the SP in order to circumvent our countermeasure.

In summary, adding the GPA to the tweak mechanism prevents a meaningful modification of the VM's memory via *AV4: SLAT Remapping*. This prohibits both our attacks against SEV's cryptographic memory isolation as well as other attacks that remap the VM's memory [60].

5.3 Summary

In this chapter, we investigated *RQ2: How can we restore cryptographic memory isolation of Trusted Execution Environments (TEEs) without memory integrity?*. For this research question, we analyzed which countermeasures could rectify the different attack vectors required for the attacks we described in the previous chapter. We determined that four of the attack vectors, namely *AV1: Page Access Tracking*, *AV2: Network Monitoring*, *AV3: Disk Access Monitoring* and *AV5: Interrupt Injection* will be challenging to prevent as they are side effects from basic virtualization tasks. Instead, we proposed a countermeasure for

Table 5.1: An overview of countermeasures proposed by related work as well as in this work and their addressed attack vectors.

Countermeasure	*AV1: Page Access Tracking*	*AV2: Network Monitoring*	*AV3: Disk Access Monitoring*	*AV4: SLAT Remapping*	*AV5: Interrupt Injection*	*AV6: Unprotected DMA Channel*	Note
Have IOMMU Encrypt Data [91]						✓	Difficult to realize
Split Hypervisor Into Different Trust Levels [154, 92]	✓			✓			Only outsources trust
Outsource Memory Management [150]	✓			✓			Only outsources trust
Full Integrity Protection [91]				✓			High overhead
Reverse Mapping Table [3]				✓			Only recently available
Reduce Information Leakage	✓[a]						Only complicates GFN identification
Purge Secrets from Memory							Not always feasible
Store Secrets in Dedicated Hardware							Does not prevent the attack itself
Avoid Copying Data from Shared to Private Memory						✓	Requires software modifications
Integrity Protection by VM				✓			High overhead
Tweak Modification				✓			Can be integrated into existing tweak

[a] Does not prevent the attack vector, but reduces its impact

AV6: Unprotected DMA Channel and focused on *AV4: SLAT Remapping*, as the elimination of this attack vector would prevent both our attacks.

Table 5.1 gives an overview of the countermeasures proposed by related work (Section 3.2) as well as in this chapter. The table shows that countermeasures previously proposed by academia either focus on moving trust away from the hypervisor to other components [154, 150, 92] or are difficult to implement [91]. Also the different countermeasures we discussed that can be applied by the VM are challenging to implement or cannot be considered a long-term solution. Hence, a change to SEV is necessary to prevent *AV4: SLAT Remapping* allowing us to circumvent its cryptographic memory isolation.

To enable such a change, we presented our contribution *C4*, a modification to SEV that prevents our attacks. In detail, we propose to include the GPA in the tweaking mechanism included in the memory encryption. We showed how this slight modification allows to prevent both our attacks against the VM's confidentiality and its integrity as well as others [60]. While we were not able to evaluate our countermeasure in practice, its practicability was shown by the latest iteration of SEV, SEV Secure Nested Paging (SEV-SNP), which is taking a similar approach. Specifically, SEV-SNP keeps track of GFN to SFN mappings, enabling it to detect and prevent remapping attacks as presented in the previous chapter.

However, also strong integrity protection cannot protect against attackers gaining control over a TEE due to vulnerabilities in the TEE's code. Therefore, in the next chapter we discuss methods to detect such attacks even in the presence of software vulnerabilities within the TEE.

CHAPTER 6

Using TEEs to Verify Run-Time Integrity

To ensure their trustworthiness, Trusted Execution Environments (TEEs) provide mechanisms to attest their configuration and the integrity of the data used for their launch [3, 34]. However, this attestation mechanism is static, as it only provides information over the TEE's state at the time of launch. In other words, the attestation does not allow to detect an attacker exploiting a vulnerability in the software running within the TEE at run-time, even if the attestation is performed after the attack.

In order to also detect run-time attacks, we analyze in this chapter how we can create a dynamic attestation process which also captures the TEE's run-time information. By doing so, we investigate our third research question, *RQ3: How can we detect the exploitation of software vulnerabilities within TEEs?*. Taking one step at a time, we first discuss how we can permit a TEE to collect run-time information of an application. For this, we use a TEE protecting userspace applications, namely Intel Software Guard Extensions (SGX), in Section 6.2. While SGX has been rumored to be deprecated, Intel only decided to no support SGX on client platforms, but continues to support server platforms [70]. Hence, the contributions we discuss in this chapter are also relevant for up-to-date platforms. In detail, we present our contribution *C5*, an architecture that allows a monitoring process to retrieve run-time information from applications. At the same time, our architecture protects both the application and the monitoring process from high privileged adversaries by hosting them in TEEs.

In the next step, we analyze how the monitoring process can use the capability to collect run-time information to detect the exploitation of software vulnerabilities in the application. To be precise, we present our contribution *C6*, in which we port Control-Flow Attestation (CFA) to TEEs. Part of *C6* is a technique that enables us to detect Data-Oriented Programming (DOP) attacks by establishing the maximum number of iterations for every loop, which we did not previously discuss in any of our publications. Our final design, which we describe in Section 6.3, permits the monitoring process in a TEE to detect clients exploiting software vulnerabilities in the application running in a different TEE. To demonstrate the practicability of our design, we describe our prototype based on Intel SGX in Section 6.4. We use this prototype to perform a detailed

performance evaluation in Microsoft Azure in Section 6.5. Finally, we discuss different aspects of our design and implementation in Section 6.6.

6.1 Attacker Model

For this chapter, we assume a cloud environment providing us with a Virtual Machine (VM) with full administrative privileges. This VM is running the service which we aim to protect: our *target*. As with most services, we have to assume that clients of the target are potentially malicious. By exploiting vulnerabilities in the target, a malicious client can gain full control over the target's memory at a specific point in time. Yet, we assume that techniques such as Data Execution Prevention [101] protect the target's code. Using Intel SGX, this is ensured in version 1 by preventing the Operating System from changing the enclave's pages permissions, and in version 2 by requiring the enclave's permission to do so [68]. However, the attacker will still have full control over the target's data memory, even if the target is running in a TEE. This is caused by the fact that exploiting the target gives the attacker access to the entire isolation domain. Having this control, the attacker can perform control data attacks [88, 14, 32] and non-control data attacks (Section 2.3).

As we are hosting the target in a cloud environment, we additionally consider the possibility of a malicious administrator controlling the software and hardware infrastructure. To gain this control, the attacker may have escaped a VM hosted on the same physical system by exploiting a vulnerability in the hypervisor [143, 136, 30, 31]. Another possibility might be that the infrastructure has been set up for malicious purposes. Either way, we deem the exact method with which the attacker gained administrative access to the infrastructure to be out of scope for this work. More importantly, having control over the infrastructure allows the malicious administrator to extract and modify data and processes within our VM. The administrator can either achieve this goal by using software-based methods, such as Virtual Machine Introspection [51], or hardware-based methods, such as DMA [11, 16] or cold boot attacks [58]. However, we do not consider the malicious administrator resorting to advanced physical attacks such as tapping buses or dismantling the CPU.

6.2 Enabling TEEs to Measure Run-Time Integrity of Applications

Having determined how to ensure the integrity of TEEs even in the presence of a malicious administrator (Chapter 5), we continue by discussing how we can use such environments to verify the run-time integrity of applications on the example of Intel SGX. Figure 6.1 depicts the generic sequence of steps required for such a verification. We assume that both the application we want to verify, called the target, and the TEE we use to verify the application, called the *VerifyTEE*, are running on the same system. While a possibly malicious client is interacting with the target (❶), the VerifyTEE monitors the target (❷). The VerifyTEE also stores the results of this monitoring process in the

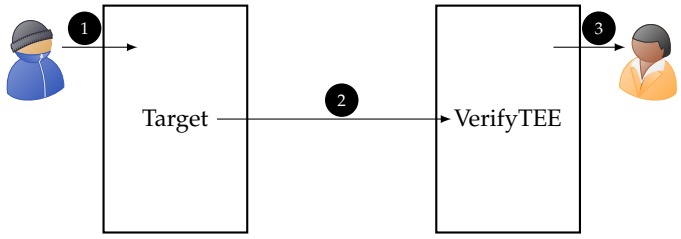

Figure 6.1: Using a TEE to verify the run-time integrity of an application. While the client interacts with the target, the VerifyTEE monitors and logs the target's state. This log can then be accessed by a trusted verifier.

attestation log. This gives a *trusted verifier*, such as the owner of the target, the possibility to retrieve information about the target's current state by fetching the attestation log (❸).

Both the first and the third step do not face major scientific challenges. Specifically, to establish a communication channel with the target (❶), the client can use any traditional mechanism such as the creation of a TLS connection. Similarly, mechanisms exist that enable TEEs to access persistent storage in order to store the attestation log [34, 82]. The persistently stored attestation log can then be retrieved by a trusted verifier (❸) via a secure, attested TLS connection [81].

However, monitoring the target from the VerifyTEE (❷) poses two major challenges. First, the VerifyTEE requires access to the target's memory or a comparable capability to collect information about the target's current state. Previous work uses ARM TrustZone [120] to monitor the integrity of an untrusted kernel from the TEE [9, 53]. Using TrustZone facilitates such a monitoring process, as it allows the TEE to access the memory of the entire system [120]. Yet, we target cloud environments, which to the best of our knowledge only provide TEEs running in user-space, such as Intel SGX or AMD Secure Encrypted Virtualization (SEV). Hence, we require a mechanism that permits such a user-space TEE to collect information about the state of the target running outside the TEE. The second challenge for the monitoring is to use this capability to collect information of the target's current state to determine its integrity. In this section, we focus on the first challenge and show how a user-space TEE can collect the required information, which is our contribution *C5*. In the next section, we capitalize on this ability by porting CFA to TEEs (Section 6.3).

One possible solution to allow the process of the VerifyTEE to collect data from the target process would be the creation of a debug bridge. Such a bridge could be created using `ptrace` [57] on Linux or the `ReadProcessMemory` [102] function on Windows. Yet, an important disadvantage of this approach is that it requires to trust the Operating System (OS). To be precise, while the OS would not be able to access the VerifyTEE, the VerifyTEE would still need support from the OS to establish the debug bridge. Although it would be possible to verify if a debug bridge was established, it would be difficult to ensure that the OS pointed the bridge to the correct target.

To reduce the dependency on the OS, we propose a design based on Intel SGX that does not rely on debug bridges. For this design, we make use of the fact that with SGX, the Host Application (HA) and the enclave share the same virtual address space [131]. We use this property to launch both the VerifyTEE and the target from the HA. Therefore, both entities will run in the same process and share the same address space. Despite the shared address space, the target and the HA will not be able to access the VerifyTEE's memory as it is protected by the TEE. However, the shared address space enables the VerifyTEE to access the target's memory without requiring any support from the OS. This permits the VerifyTEE to collect run-time information from the target.

A problem that remains with this approach is that the target itself is not protected by a TEE and hence exposed to high privileged adversaries. While monitoring by the VerifyTEE would allow to detect integrity violations of the target, it would be difficult to ensure its confidentiality. For example, a malicious administrator could still extract secrets located in the target's memory. To protect against such a threat, we propose to also host the target in a TEE, protecting it from high privileged adversaries. Specifically, we use the same HA to launch two TEEs: the *ProveTEE* hosting the target, and the VerifyTEE responsible for ensuring the target's integrity. This permits us to protect both the verification mechanism and the target from a malicious administrator. Yet, a difficulty that arises with this design is that the VerifyTEE is no longer able to access the target's memory, as it is now protected by a TEE. To work around this limitation, we make use of an agent in the ProveTEE, which we call the *trampoline*. The trampoline is responsible for collecting information from the target and providing it to the VerifyTEE. For the latter, we again make use of the fact that as both TEEs are launched by the same HA, they share the same address space. Therefore, we can establish a shared memory region between both TEEs which we use to exchange data.

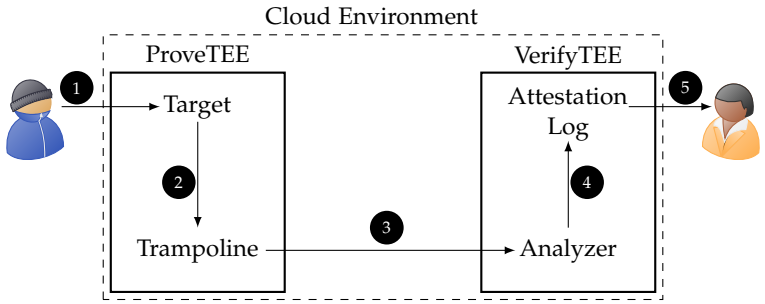

Figure 6.2: Overview of our design for *C5*. One of its two goals is to prevent a malicious administrator from extracting or modifying critical data, which we aim to achieve by using two separate TEEs. This separation also helps us to achieve our second goal, ensuring that a malicious client cannot influence monitoring results by compromising the target or the ProveTEE. Source: [108].

Figure 6.2 shows our final design aiming to prevent a malicious administrator from accessing the target and a malicious client from influencing monitoring results. The

dashed line indicates the cloud environment offered by the cloud provider. Within this environment, we create two TEEs that enable us to protect our processes from a malicious administrator, the ProveTEE and the VerifyTEE. The ProveTEE is responsible for hosting the target, which communicates with possibly malicious clients (❶). Furthermore, it contains the trampoline, responsible for collecting control-flow information from the target before the flow gets executed (❷). In comparison, the VerifyTEE hosts the *analyzer*, responsible for verifying the data collected by the trampoline (❸). By outsourcing the verification into a separate TEE, we aim to protect this step from a malicious client compromising the ProveTEE. After verification, the analyzer stores the result in the attestation log also located in the VerifyTEE (❹). The VerifyTEE itself is entirely passive: its tasks are the analysis of monitoring data received from the trampoline and the provision of the attestation log to the trusted verifier (❺). Furthermore, the VerifyTEE may be equipped with additional functionalities such as deploying target-specific mitigations or shutting down the target and its service, which are out of the scope of this work.

In the following sections, we use this approach to port CFA to TEEs. Specifically, in Sections 6.3 and 6.4, we describe our design and its implementation in detail. Afterwards, we evaluate our implementation in Section 6.5 and analyze if we were able to achieve our goals by discussing possible limitations in Section 6.6.

6.3 Porting Control-Flow Attestation to TEEs

In the previous section, we proposed a design which allows an analyzer running in the VerifyTEE to monitor the state of a target in the ProveTEE. Now, we continue with discussing how we can use this ability to verify the target's integrity at run-time. To be precise, we make use of CFA to perform the verification (Section 2.3). The concept of CFA was first proposed by Abera et al. [1] in order to detect control data attacks. CFA records the control flow of a program and afterwards compares it to a set of previously determined legal control flows. While this approach permits to detect control data attacks, recent work additionally aims to detect non-control data attacks [38, 139]. However, most previous work targets embedded systems, focusing on specific challenges for such environments, such as limited resources. In comparison, cloud environments have access to a vast amount of resources, while at the same time presenting different challenges, such as a contrasting usage model. Specifically, while embedded systems are mainly used by their owners, cloud applications are often designed to provide a service to autonomous clients. Those clients are often considered untrusted. Hence, performing CFA in cloud environments requires us to move the attestation procedure away from the client to other, trusted entities. It is exactly this gap, the absence of CFA designs adapted to cloud environments, which we aim to close in this section. In detail, we present our contribution *C6*, a design in which we port CFA to TEEs.

The main goal of our design is to detect control data and DOP attacks (Section 2.3) in untrusted cloud environments, posing several challenges to overcome. While pre-

vious work on CFA mainly focused on embedded systems, our design targets cloud environments. Although such an environment reduces challenges such as the need to save resources, new requirements and challenges arise, such as a possibly malicious administrator. In this section, we identify those challenges and present our respective solutions that form our design. After giving an overview of our design porting CFA to TEEs, we continue with explaining the details of its two phases, the offline and the online phase.

6.3.1 Overview

Most applications that run in cloud environments and communicate with clients represent some kind of service, such as a signing service for health certificates. These services are, for example, deployed within the European Union to sign the so-called Digital Green Certificates providing proof of vaccination against or recovery from COVID-19 or a negative test result [45]. Such a signing service receives the hash of a newly created health certificate, signs the hash with its private key, and returns the signature. To avoid the forging of health certificates, it is important that the service's private key is equally protected from a malicious administrator and a malicious client. Our design protects the target signing service against both types of adversaries: By using TEEs, we prevent an administrator from accessing the service's private key. Additionally, we can detect a malicious client compromising the service. In the event of such a compromise, the service owner can use the collected information to identify and resolve the vulnerability. Then, by installing a new private key and revoking health certificates signed with the old key, the impact of the attack can be minimized. To achieve this protection, we only require the developer to annotate the start and the end of the CFA, but do not require any other changes to the target.

Another relevant aspect of signing services is that they only sign newly created health certificates and are therefore only required to provide a limited throughput. In other words, their performance is of much lower priority than their security guarantees, which could be threatened by a malicious client. To protect against such a threat, we move the attestation away from the client to another, trusted entity. To be precise, previous designs for CFA require the client to attest the remote execution [1, 39, 153, 141]. In comparison, we assign this task to a trusted verifier such as the owner of the target, thereby separating the role of the client from the role of the trusted verifier. This separation serves three purposes: First, it permits to leave the interface between the client and the service unchanged, making our design transparent to the client. Second, it allows us to perform the attestation time-independent from the communication between client and service. Third, it enables us to perform CFA without requiring any information from the client, as done for example by previous work to ensure the freshness of the attestation [1, 39, 153, 141, 2, 33, 139].

An additional concern in cloud environments is that we cannot rely on the underlying infrastructure, as it might be controlled by a malicious administrator. To protect against this threat, we place both the target and the verifier in a TEE, shielding them from

high privileged adversaries. By using two different TEEs, we limit the influence of a malicious client on the verifier after exploiting vulnerabilities in the target. This prevents an adversary in control of the target from modifying attestation information as it can be done for Control-Flow Integrity (CFI) [52, 54, 115].

When choosing a TEE, we have to consider that cloud environments are limited in the choice of available hardware. Furthermore, we are not able to perform any hardware modifications, requiring us to use the TEE offered by the cloud provider. Hence, we start by discussing how to port CFA to TEEs based on the generic concepts of TEEs, preserving the flexibility of which TEE to choose. In Section 6.4, we build on the architecture we established in Section 6.2 and show how our design can be used on the example of Intel SGX. However, future work could also implement our design with other TEEs, for example by using the privilege layers introduced in SEV Secure Nested Paging (SEV-SNP) [3]. With discuss this possibility in more detail in Section 7.2.

Our design comprises two phases, the offline and the online phase. In the offline phase, the analyzer learns the target's control-flow information. This information contains the target's Control-Flow Graph (CFG) which we use to detect control data attacks. Furthermore, it contains the maximum number of iterations for each loop, which we call the *iteration limit*, with which we detect DOP attacks (Section 2.3). The subsequent online phase builds on the five steps we established in the previous section (Section 6.2), adapted to CFA. In detail, the client sends regular requests to the target (❶), making our design transparent to the client. While the target is processing the request, the trampoline records the control-flow information and iteration limit of each loop (❷). Next, the trampoline forwards the information to the analyzer in the VerifyTEE (❸). The forwarding prevents an attacker in control of the target, or even the entire ProveTEE, from tampering with the control-flow analysis. To be precise, it ensures that once the VerifyTEE received the information, it cannot be modified by a compromised ProveTEE. The analyzer then uses the control-flow information provided by the trampoline to compare the collected control flow against the target's CFG. Additionally, it verifies for each loop that the number of iterations does not exceeded its iteration limit. Next, the analyzer stores the result of both checks in the attestation log (❹). Finally, the trusted verifier can retrieve the attestation log from the VerifyTEE (❺).

6.3.2 Offline Phase

Before being able to verify the target's control flows in the online phase, we require an offline phase to prepare the environment. For this preparation, we instrument the target to regularly pass control to the trampoline. This instrumentation enables the trampoline to record the target's executed edges and loop iteration information (❷) and to provide this information to the analyzer (❸). As we perform the instrumentation at compile-time, we do not require any high-level modifications. Instead, the developer only needs to annotate where the attestation of the target's control flow should start and end. This makes our instrumentation transparent to the target's clients, as the interface between the client and the target remains unchanged.

Having established a mechanism that allows the trampoline to record the target's control flow, we calculate the target's CFG via dynamic execution. Using this approach, we send requests to the target until it has executed every control flow path at least once. To ensure that no malicious paths are followed, we perform this step in a safe environment without any possibly malicious client interaction. During the dynamic execution, the analyzer collects all recorded control flows and merges them into the CFG. Additionally, it determines the iteration limit for each loop.

Yet, determining the point at which a complex target has executed all legal control flows is still an open research question. Therefore, our design aims to protect targets with reduced complexity frequently deployed in cloud environments, the so-called microservices [6]. Such microservices split the functionality of a complex multi-functional service into multiple, simple services with a single functionality. This reduced functionality also reduces the complexity of each service, simplifying the execution of all legal control flows for each of them. Alternatively to splitting up the target itself, we can break down the CFG of a complex target into multiple segments in order to facilitate the collection of each segment's CFG [2, 141].

Another difficulty for the creation of the CFG is that most OSs deploy Address Space Layout Randomization (ASLR) [135]. ASLR causes the target to use different virtual addresses at every launch, preventing us from using addresses to identify the endpoints of edges in the CFG. Instead, we identify the endpoints by assigning them unique identifiers, the *CFG IDs*, leaving the identification unaffected by ever-changing virtual addresses.

In order to also detect DOP attacks, we make use of the fact that they require an increased number of loop iterations during the attack (Section 2.3). To be precise, for our loop verification, we assign a unique identifier to every loop, the *loop ID*. We use this loop ID in the offline phase to assign an iteration limit to each loop. Afterwards, in the online phase, we transfer the loop ID to the analyzer each time the target enters a loop and for each subsequent iteration of the loop. This information is sufficient for the analyzer to detect when a loop exceeds its iteration limit determined in the offline phase, indicating a DOP attack.

Figure 6.3 shows an example CFG for a signing service, which we annotated with CFG IDs and loop IDs. The services reads input data in a loop between N_1 and N_2. This step requires two different edges, to the endpoints of which we assigned the CFG IDs 1 – 4. For the loop, we assigned the loop ID 1. Each time the service executes the edge between N_2 and N_1, we count the number of loop iterations and store the maximum as iteration limit for the respective loop. For example, let us assume that the maximum input length the server processes is 256 bit, or 32 byte. In case each execution of N_1 reads one byte, the service requires a maximum of 32 loop iterations to read the entire input. Hence, we would determine an iteration limit of 32 for loop ID 1. Also note that both endpoints starting from N_2 have the same CFG ID. This is due to the fact that both have the same starting point, which is the loop condition in N_2. When the loop is finished, the service continues to N_3, in which it checks if the input has the expected

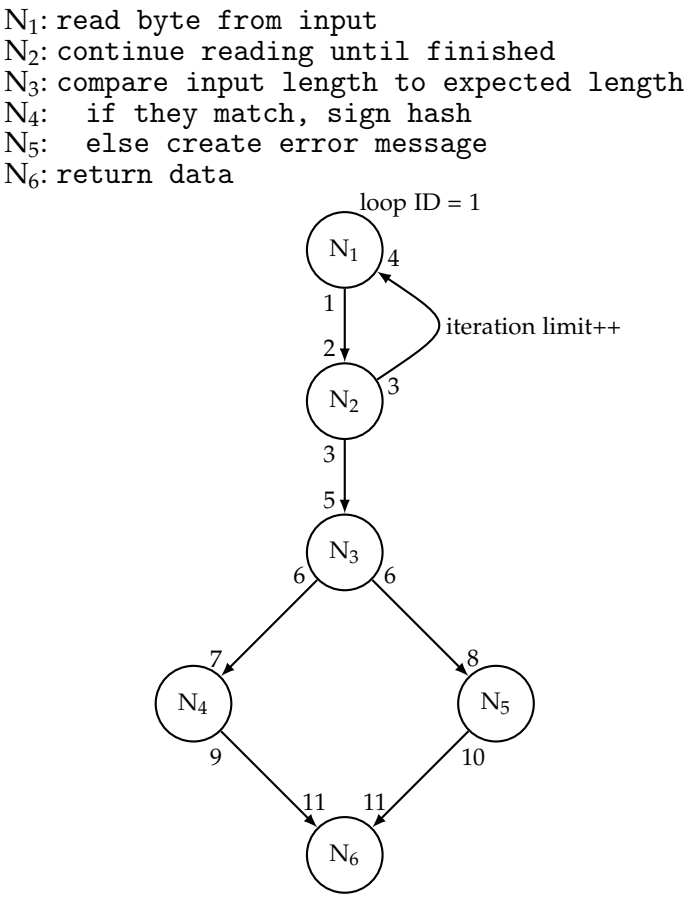

Figure 6.3: CFG annotation in the offline phase. For the endpoint of the edges, we assign the CFG IDs 1 – 11. Additionally, we assign a loop ID to each loop and calculate its iteration limit.

length. Depending on the result, the service continues either to N_4 or N_5. As with N_2, both endpoints starting from N_3 have the same starting point, namely the `if` condition. Hence, they also share the same CFG ID. After N_3, the service either signs the received hash in N_5 or creates an error message in N_4. Finally, the data is returned in N_6. Again, both endpoints leading to N_6 have the same CFG ID, namely 11. The reason for this is that both edges lead to the same destination.

By using the CFG IDs, we can determine all valid edges and calculate the target's full CFG in the offline phase. Additionally, we are able to determine the iteration limit for each loop with the help of the loop IDs. In combination, this allows us to detect control data attacks and DOP attacks. Furthermore, having access to the full CFG, the analyzer can also identify the node in which the compromise has first shown effect.

6.3.3 Online Phase

In the subsequent online phase, we perform CFA and loop verification while the target processes requests from potentially malicious clients. During the online phase, the target hosted in the ProveTEE waits for requests from clients (❶). For the client, the additional protections are transparent. Therefore, clients can send the same requests to the instrumented target as to the original.

While the target processes the request, the trampoline records all CFG IDs and loop IDs (❷). However, the trampoline itself is not responsible for processing the IDs. This is due to the fact that it runs in the ProveTEE, next to the target. Therefore, an attacker gaining control over the target could also tamper with the trampoline. To ensure that such an attack influences neither the CFA nor the loop verification, we shift these analysis tasks to the VerifyTEE. Hence, the only task of the trampoline is to forward the collected IDs to the analyzer in the VerifyTEE (❸). Aside from protecting the analysis, another advantage of shifting this step to a different TEE is that it reduces the processing time of the trampoline, as it only forwards the IDs.

Next, the analyzer uses the received CFG IDs to reconstruct the control flow of the target. By comparing the reconstructed control flow against the target's CFG created in the offline phase (Section 6.3.2), the analyzer can detect control data attacks. Specifically, if two recorded CFG ID do not correspond to an edge also taken in the offline phase, we can assume that an attacker was able to hijack the target's control flow. Additionally to the CFA, the analyzer uses the loop IDs to perform loop verification. To be precise, it counts the number of loop iterations and compares them to the iteration limit determined in the offline phase (Section 6.3.2). This allows the analyzer to detect DOP attacks increasing the number of iterations for a loop. Having finished both the CFA and the loop verification, the analyzer stores detected violations in the attestation log (❹).

Finally, the VerifyTEE provides the trusted verifier with the contents of the attestation log (❺), containing all requests that altered the target's control flow. In the event of such a request, the owner can then take appropriate actions, such as analyzing and resolving bugs in the target's code.

6.4 Implementation

Based on our design, we implemented a prototype using Intel SGX (Section 2.2) as TEE[1][2], which we discuss in this section. We start by giving on overview of our prototype and describing how we exchange data between the ProveTEE and VerifyTEE (Section 6.4.1). Afterwards, we describe our additions to the LLVM compiler framework which permit us to instrument the target to regularly pass control to the trampoline (Section 6.4.2). Finally, we discuss the tasks of the two threads among which we split up the work (Section 6.4.3).

[1] `https://github.com/Fraunhofer-AISEC/guarantee-code`
[2] `https://github.com/Fraunhofer-AISEC/guarantee-llvm`

6.4.1 Overview

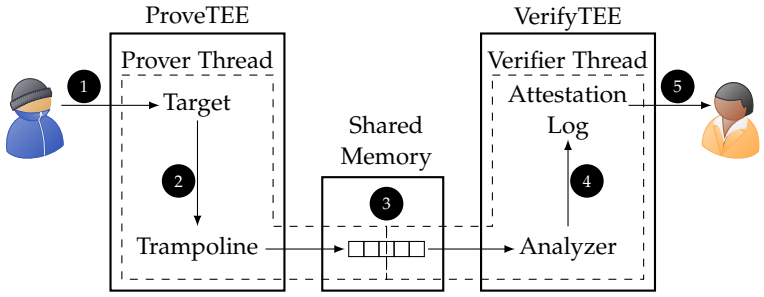

Figure 6.4: Overview of our prototype for *C6*. The shared memory region allows to transfer the IDs collected by the trampoline to the analyzer in the VerifyTEE. Furthermore, we distribute the work between two different threads, the prover thread and the verifier thread.

Figure 6.4 gives an overview of our implementation. The basic building blocks are the same as in Figure 6.1. However, we introduced two threads between which we divide the work, the *prover thread* and the *verifier thread*. We discuss the tasks of both threads in detail in Section 6.4.3. Furthermore, we added a shared memory region used to exchange data between the ProveTEE and the VerifyTEE. This region permits the TEEs to exchange information without requiring any context switches, thereby reducing the performance overhead of the communication.

To create the shared memory region, we make use of the fact that SGX TEEs share the virtual address space with their Host Application (HA) [131]. Therefore, by using the same HA to launch both the ProveTEE and the VerifyTEE, both TEEs share the same virtual address space [105]. This enables us to allocate a memory region in the HA which both the ProveTEE and the VerifyTEE can access, allowing them to exchange collected IDs. Note that while both TEEs are able to read and write the shared memory, they are not able to access each other's memory.

The first step of the prototype is to perform an offline phase, in which it collects all information necessary to perform CFA and loop verification (Section 6.3.2). Afterwards, in the online phase, it verifies the target's control flow and iteration limit of each loop and provides the verification results to the trusted verifier (Section 6.3.3).

In our implementation, each node in the CFG corresponds to a basic block in the target's code. A basic block is a code section with a single entry and a single exit [95]. The basic blocks are connected via edges, the endpoints of which are recorded by the trampoline. To be precise, the target calls the trampoline each time before entering or exiting a basic block, providing the CFG ID of the respective endpoint. Similarly, the trampoline also records the loop IDs for every entry and subsequent iteration of the loop. On execution of the trampoline, the ID is stored in an *ID batch*, which acts as a cache. When the ID batch is full, the trampoline encrypts and stores it in the *queue* located in the shared memory region.

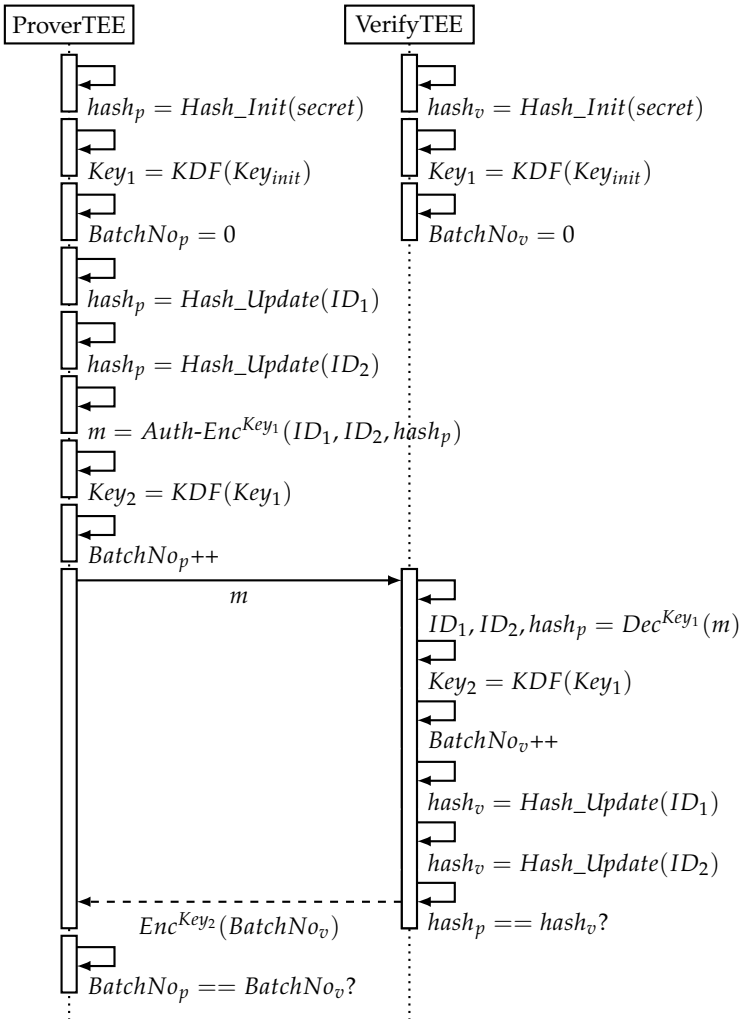

Figure 6.5: The exchange of IDs between the ProveTEE and the VerifyTEE. To ensure the integrity of IDs cached in the ID batches, we use a hash chain. Additionally, we calculate a new encryption key for every exchange with the help of a KDF and record the number of the current batch. Source: [108].

Figure 6.5 depicts how we can securely cache the IDs by combining the calculation of a new key for every data exchange with the tracking of the number of exchanged ID batches. To ensure the integrity of the cached IDs in the ProveTEE, we make use of a hash chain. The root of this hash chain is a secret the service owner provides to both the ProveTEE and the VerifyTEE. When the trampoline caches an ID in the ID batch, we update the hash in the ProveTEE, $hash_p$, with the new ID. If the ID batch is full, we combine it with the current value of $hash_p$ to form a single message, which we then encrypt and store in the *queue* located in the shared memory region. After the VerifyTEE reads and decrypts the message from the queue, it adds all IDs to its own hash, $hash_v$. As we initialize both $hash_p$ and $hash_v$ with the same secret and add the same IDs in the

same order to both hashes, $hash_v$ should equal the value of $hash_p$ transmitted with the batch. To minimize the performance impact of this step, our prototype uses BLAKE3 as hash algorithm [113].

When exchanging the ID batch via shared memory, we have to keep in mind that the memory is also accessible to other, untrusted entities such as the HA or the OS. This means that a malicious administrator could inspect and modify transferred ID batches. To protect the batches from such attacks, we implement mechanisms which ensure their confidentiality and integrity. For confidentiality, we provide both TEEs with an initial encryption key, Key_{Init}. Both TEEs use this initial key as input to a Key Derivation Function (KDF) in order to calculate an encryption key. For the following messages, the encryption key serves as input for the KDF to produce a new key for the next message, allowing us to calculate a different key for each message. These keys enable us to encrypt the ID batches and the respective hash values before writing them into shared memory, ensuring that they cannot be inspected by a malicious administrator. By additionally calculating a new key for every message, we ensure that an attacker in control of the ProveTEE is not able to infer previously used keys. We achieve this by calculating the keys with an irreversible KDF and by deleting old keys after usage. To be precise, we again make use of BLAKE3, as it is irreversible and already available in our prototype.

To ensure the integrity of the communication, we use AES-GCM for encryption, which additionally provides integrity protection [97]. AES-GCM requires an Initialization Vector (IV), which we need to synchronize between the trampoline and the analyzer to ensure correct encryption and decryption. We achieve this synchronization by managing an independent 64bit counter, *BatchNo*, in both the ProveTEE and the VerifyTEE. Note that for IVs with a different size than 96bit, AES-GCM hashes the IV to create a 96bit input [97]. Hence, using a 64bit counter does not impact the security of the encryption.

After each encryption or decryption, we increase the respective counter to keep both counters synchronized. These counters permit us to detect attacks in which the malicious administrator prevents forwarding of the ID batches. For this detection, the VerifyTEE regularly acknowledges the receipt of the ID batches. On the other side, the ProveTEE, having stored a certain number of ID batches into the queue, will wait for an acknowledgment before continuing to execute the target. We define the frequency of these acknowledgments as the *feedback frequency*. In Section 6.5, we analyze the performance and security impact of different feedback frequencies. Also note that the feedback from the VerifyTEE to the ProveTEE can cause IVs to be used twice: Once for the transfer of m from the ProveTEE to the VerifyTEE, and a second time to encrypt the feedback. However, these two transfers will use different keys, preventing the reuse of the same combination of IV and key.

6.4.2 Instrumentation

To collect control-flow information from the target, we require it to regularly pass control to the trampoline before executing control flows. We achieve this by automatically instrumenting the target during the compilation process. For this instrumentation, we

rely on the LLVM compiler framework[3]. The instrumentation serves two purposes: calculation of the target's CFG to detect control data attacks and loop verification to detect DOP attacks (Section 6.3.2).

CFG Calculation To calculate the CFG, we need to determine all edges executed by the target. Each edge has two endpoints: the exit from a basic block, and the entry into another basic block, both of which we identify with a unique CFG ID. By analyzing the recorded CFG IDs, we are able to reconstruct all edges executed by the target and therefore its full control flow. To collect the CFG IDs, we add calls to the trampoline on every entry and exit of each basic block. For each of these calls, we assign a unique CFG ID, which we store directly in the binary of the target. Furthermore, we add code that reads the ID and stores it in the general purpose register `R11` [67], from where the trampoline can consume it. We chose the `R11` register as it is caller-saved and therefore ensures that we are not overwriting function parameters.

To instrument all entries and exits, we modify two different phases of the LLVM compilation process: First, we add a pass to the Intermediate Representation (IR) optimization used for mid-level analysis and transformations [19]. In the pass, we add calls to the trampoline on every direct branch. Specifically, we add calls at the beginning and the end of each basic block and before and after direct function calls. To assign a unique CFG ID for each endpoint, we make use of a counter.

Furthermore, we add a pass to the backend that creates the x86-specific code [19] to instrument all indirect branches. Specifically, we add calls to the trampoline before and after every indirect function call, before every indirect jump, and before every return instruction. As before, each trampoline call is associated with a unique CFG ID, allowing us to identify the endpoints of all executed edges in the target's control flow. Additionally, on indirect calls, indirect jumps, and returns we XOR the CFG ID with the offset between the current instruction pointer and the jump destination to detect modifications of the jump address. This safeguard permits us to detect jumps from instrumented into uninstrumented code. Specifically, it causes a modification of the jump destination to also change the recorded CFG ID, enabling us to detect the modification even if it points the execution to uninstrumented code. Being able to record all control flows, we use the offline phase to combine the collected CFG IDs to the target's CFG.

While it may technically be possible to perform all instrumentations in IR only, the backend pass permits us to precisely place the calls to the trampoline. For example, let us consider Return-Oriented Programming (ROP) attacks in which each gadget consists of the last instructions before a return instruction [132]. To detect such attacks, we need to place the call to the trampoline as close to the return instruction as possible. When adding the call in IR, the subsequent code generation in the backend will insert instructions for stack cleanup between the call and the return instruction. This would allow these instructions to be executed without invoking the trampoline, providing a potential ROP gadget which we could not detect. In comparison, using our backend

[3] `https://llvm.org/`

pass, inserting the call during code generation enables us to place the call as close to the return instruction, or any other indirect branch instruction, as possible. Note that we do not consider the instructions before direct function calls as possible ROP gadgets due to the hard-coded function address. Therefore, we refrain from instrumenting direct function calls in the backend pass and instead instrument them in our IR pass.

Loop Verification The second purpose of the instrumentation is to determine the iteration limit for each loop. For this, our prototype can apply three different methods: Using annotations provided by the developer, analyzing the canonical induction variable, or dynamic learning. For all three methods, we first assign each loop a unique loop ID in an IR pass. The loop ID allows the analyzer in the offline phase to assign an iteration limit for every loop. Additionally, the analyzer can use the loop ID in the online phase to compare the number of iterations with the previously determined iteration limit.

To determine the loop's iteration limit in the offline phase, we first analyze if the developer annotated the loop in the source code. The annotations give the developer three possibilities: (i) Statically provide the loop's iteration limit, (ii) indicate that the iteration limit should be dynamically learned, or (iii) indicate that the loop should be ignored. In case the developer statically provided the iteration limit, we use the provided information and the loop ID as parameters in a call to the trampoline, which we add to the loop's preheader [94]. The call in the preheader ensures that the target executes the trampoline each time it enters the loop. We use this call in the offline phase to provide the analyzer with information about the iteration limit of the upcoming loop. In comparison, when the developer indicated that the iteration limit should be learned dynamically, the call in the preheader only provides the loop ID, triggering the analyzer to learn the iteration limit in the offline phase. To enable the analyzer to learn the iteration limit, we add a call to the trampoline to the loop's backedge [94], which is executed after each loop iteration. This permits the analyzer to count the number of iterations for the loop and to store the highest value as the loop's iteration limit. In comparison, when the developer indicated that the loop should be ignored, we refrain from adding a call to the loop's preheader. This will cause the analyzer not to verify the loop's iteration limit, which can be, for example, useful for infinite loops performing an unknown number of iterations.

When the developer did not provide any annotations, we first attempt to determine the iteration limit by extracting the maximum value of the canonical induction variable [94]. If this extraction fails, for example due to the lack of such a variable, our prototype defaults to dynamically learning the loop's iteration limit as if annotated by the developer. This strategy liberates the developer from having to annotate every single loop. Instead, the complementary annotation allows fine-grained control of the automated determination of the iteration limit. By combining the automatic determination and the additional control via annotations, we reduce the effort required to adopt our design. At the same time, we give the developer the possibility to manually configure the loop verification for specific loops if desired.

Note that while our prototype is capable of different approaches to determine the iteration limit in the offline phase, the loop verification in the online phase always follows the same procedure. Specifically, in the online phase we compare the number of loop iterations with the iteration limit determined in the offline phase. To count the number of iterations, we reuse the instrumentation for dynamically learning the iteration limit in the offline phase. In detail, we call the trampoline in the loop's backedge, notifying the analyzer about every iteration.

6.4.3 Execution

Having discussed how we instrument the target, we continue by describing how we utilize the instrumentation during the execution of our prototype. Specifically, the instrumentation allows us to collect the target's valid control flows in the offline phase. Furthermore, it permits us to verify the target's control flows in the online phase by comparing them to the valid control flows collected in the offline phase. To minimize the impact of this additional verification, we split the tasks into two threads, the prover thread and the verifier thread, which we also describe in this section.

Our automatic instrumentation (Section 6.4.2) causes the target to regularly pass control to the trampoline. This enables the trampoline to collect information about the target's control flows and to forward this information to the analyzer. In the offline phase, we use this ability to collect all of the target's valid control flows via dynamic execution. To be precise, we created a script which performs a variety of requests[4], causing the target to execute each valid control flow at least once. During processing of these requests, the analyzer investigates the CFG IDs and calculates the target's CFG. Additionally, it stores the iteration limit of each loop, which we determine either statically or dynamically (Section 6.4.2). Having prepared all the required information, we switch to the online phase, in which the analyzer uses the collected control-flow information to detect attacks.

In the online phase, our prototype performs CFA and loop verification for all control flows between a start and end point annotated by the developer. Those points could be for example the acceptance and the termination of an incoming connection. The annotations at the start and end point cause the trampoline to send a special begin or end tag to the analyzer, triggering it to start or stop the analysis. These additional analysis tasks unavoidably slow down the processing time of a request. To minimize this impact, we make use of the high processing power of cloud systems. Specifically, we provision our VM with multiple vCPUs, thus supporting parallel processing of different tasks. In order to fully capitalize on this ability, we distribute the prototype's tasks between two threads: the prover thread and the verifier thread.

The prover thread is responsible for executing the target to process incoming requests. Additionally, it executes the trampoline to collect the IDs in batches and to store them in the shared queue. In detail, within the prover thread, the target waits for and processes

[4] `https://github.com/Fraunhofer-AISEC/guarantee-code/blob/master/GuaranTEE/server_learn.sh`

incoming requests (❶). During processing, the thread jumps to the trampoline on every instrumentation (❷), which caches the recorded CFG ID or loop ID in the ID batch. When an ID batch is full, the trampoline encrypts it with AES-GCM, using the key retrieved with the KDF (Section 6.4.1). Afterwards, it stores the encrypted ID batch in the queue (❸). When the trampoline finished execution, the prover thread returns to the target to continue processing the request.

In comparison, the verifier thread is responsible for the reconstruction of the target's control flow. It reads the IDs provided by the prover thread, validates them against the control-flow information learned in the offline phase, and stores the validation result in the attestation log. To fulfill these tasks, the verifier thread reads the encrypted ID batches from the queue (❸) and decrypts them with the key derived with the KDF (Section 6.4.1). Next, it forwards the CFG IDs and loop IDs contained in the ID batches to the analyzer. Using the CFG IDs, the analyzer rebuilds the target's control flow and verifies it against the target's CFG. Additionally, the analyzer uses the loop IDs to perform loop verification (Section 6.4.2). After the analyzer finished the control flow analysis and loop verification, the verifier thread stores detected violations in the attestation log (❹). Afterwards, it checks if new ID batches exist in the queue to continue the attestation.

The most efficient method to synchronize the two threads would be a mutex [111]. Unfortunately, SGX does not yet support the use of mutexes to synchronize multiple threads running in different enclaves. Instead, our current implementation has to rely on atomic operations [49] to add or read ID batches to or from the queue. Additionally, we use a spinlock [144] while the verifier thread waits for the prover thread to add new ID batches to an empty queue.

In order to retrieve the analysis and verification results (❺), our prototype allows the trusted verifier to access the attestation log in a protected manner. For this access, the verifier confirms the integrity of the VerifyTEE using the static remote attestation processes of SGX [7, 73, 128]. During this process, the trusted verifier also establishes a TLS connection, thereby creating an encrypted connection directly into the VerifyTEE [81]. Using this encrypted connection, the verifier can securely retrieve the attestation log.

To summarize, the analyzer uses the offline phase to calculate the target's CFG and to collect the iteration limit for each loop. In the online phase, it verifies the target's recorded control flow by comparing the CFG IDs with the previously constructed CFG. Additionally, the analyzer performs loop verification by comparing the recorded loop iterations with the iteration limit determined in the offline phase. To reduce the impact of the additional analysis, we split the tasks between the two threads, the prover thread and the verifier thread. While the prover thread executes the target and collects its control-flow information, the verifier thread performs the analysis of the collected information.

6.5 Evaluation

In order to analyze the overhead of our prototype, we performed a detailed performance evaluation in a cloud environment. After introducing our setup (Section 6.5.1), we first discuss the performance of our prototype by using a benchmark (Section 6.5.2). The benchmark evaluates the performance under extreme conditions, such as a very high CPU load. In comparison, a regular target, such as a signing service, will also perform less resource-intensive operations such as waiting for the OS or for incoming network traffic. This will reduce the overhead of a regular target in comparison to the evaluated benchmark. To prove this claim, we continue by analyzing the overhead with a regular target by using the example of our signing service. With the signing service, we first evaluate the processing times of the different components (❷ – ❹) separately (Section 6.5.3). Additionally, we use the service to perform a detailed performance evaluation of the entire prototype (Section 6.5.4).

6.5.1 Setup

As target for the evaluation, we created a microservice responsible for signing health certificates[5] (Section 6.3.1). Our signing service receives the SHA256 hash of a newly created health certificate, signs the hash with its private RSA 2048bit key, and returns the signature. The service performs this exchange via a secure connection, which we establish based on code from the SGX-OpenSSL project[6]. This code allows clients to establish a TLS channel directly into the TEE. By using annotations, we instructed our prototype to perform the CFA and loop verification between accepting and terminating an incoming connection. In total, our signature service consisted of 4,533 instructions, excluding libraries such as OpenSSL, and the trampoline of 47,542 instructions. To instrument the target, we compiled it with LLVM version `11.0.0` and our additional IR and backend pass[7].

As traditional time measurements are not available within SGX TEEs [18], we added an ocall to both TEEs which notifies the HA to start or stop a measurement. By executing this ocall at the start and at the end of our measurements, we were able to determine the difference between these two points in time in the HA.

To perform the measurements, we chose Microsoft Azure as cloud environment, which provides users the possibility to run SGX within VMs [126]. For the VM, we chose a `Standard DC4s_v2` machine running on an `Intel Xeon E-2288G` CPU, providing four vCPUs and 16 GiB of memory. Within the VM, we were running the default Ubuntu 18.04 provided by Azure with kernel version `5.4.0-1089-azure` and the Linux SGX DCAP driver in version `v1.41`.

[5] `https://github.com/Fraunhofer-AISEC/guarantee-code`
[6] `https://github.com/sparkly9399/SGX-OpenSSL`
[7] `https://github.com/Fraunhofer-AISEC/guarantee-llvm`

6.5.2 Benchmark Performance Evaluation

To determine the throughput of our prototype under high load, we started by deploying a benchmark as the target before evaluating the performance of our signing service. To be precise, we made use of the `sgx-nbench` benchmark[8], which is a port of `nbench` [117] to SGX. As a baseline, we first ran the benchmark without instrumentation. Then, we instrumented the target and configured the default values of an ID batch size of 10,000 and a feedback frequency of 10. In Sections 6.5.3 and 6.5.4, we analyze the impact of different batch sizes and feedback frequencies.

Table 6.1: Comparison of `sgx-nbench` with and without instrumentation. While the second column shows the number of iterations per second with the uninstrumented `sgx-nbench`, the third and fourth column show the number of iterations with instrumentation and the respective overhead.

Test	**Uninstrumented**	**Instrumented**	**Overhead**
Numeric Sort	2,387.2	9.9	240x
String Sort	1,285.1	29.6	42x
Bitfield	$9.3382 * 10^8$	$2.484 * 10^6$	375x
FP Emulation	1,251	10	124x
Fourier	62,574	10,822	5x
Assignment	121.51	0.15	809x
Idea	21,500	52	412x
Huffman	6,509.4	30.4	213x
Neural Net	192.77	1.62	118x
LU Decomposition	4302.4	15.51	276x

Table 6.1 shows the results of both the uninstrumented `sgx-nbench` target and the instrumented version with its respective overhead. This overhead varies between 5x for the Fourier Test and 809x for the Assignment Test. On average, we recorded an overhead of 261x between the uninstrumented and the instrumented benchmark. Having determined this relatively high overhead, it is important to note that a benchmark tests the performance under extreme conditions. To prove that our the overhead of our prototype drastically decreases with traditional targets, we continue the evaluation by using our signing service as target. With this target, we evaluate the different components separately as well as the overhead of the entire prototype.

6.5.3 Component Performance Evaluation

Using our signing service (Section 6.5.1), we continued by evaluating the different components, namely the instrumented target (❷), the queue (❸), and the analyzer (❹). To evaluate the throughput of the instrumented target (❷), we simulated 100,000 client

[8] `https://github.com/utds3lab/sgx-nbench`

requests. While processing the requests, the target produced 11.56 IDs per microsecond, or, in other words, called the trampoline on average every 86.53 ns. We call this frequency with which the target calls the trampoline the *trampoline call frequency*.

Next, we determined the throughput of the queue used to transfer IDs from the trampoline to the analyzer (❸). For this transfer, the trampoline caches the received IDs in batches (Section 6.4.1). When an ID batch is full, the trampoline encrypts and stores it into the queue, where the analyzer can read it. Having received and successfully decrypted the ID batch, the analyzer gives feedback to the trampoline. Determined by the feedback frequency (Section 6.4.1), the analyzer can be configured to give feedback only after a certain amount of ID batches has been processed. Using this approach, the throughput with which we can transfer the IDs from the trampoline to the analyzer depends on two variables: the ID batch size and the feedback frequency.

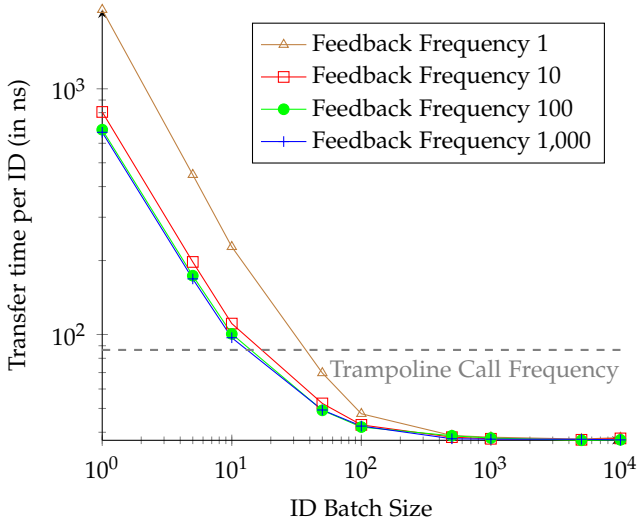

Figure 6.6: Average time required to transfer a single ID from the trampoline to the analyzer. While the impact of the feedback frequency is limited, the transfer time drastically decreases with higher batch sizes.

Figure 6.6 gives an overview of the throughput with different ID batch sizes and feedback frequencies. While the X-axis depicts the different batch sizes, the Y-axis indicates the average transfer time per ID in nanoseconds. Both axes are in logarithmic scale. The four different plots indicate the impact of different feedback frequencies on the transfer time. A feedback frequency of 1 means that the trampoline waits for feedback from the analyzer after every single ID batch. In comparison, using a frequency of 1,000, the trampoline only waits for feedback after having transferred 1,000 ID batches.

The four different plots show that the feedback frequency has only limited impact on the throughput. While a frequency of 1 does cause a higher performance overhead, the differences between the frequencies 10, 100, and 1,000 are only minor. This difference becomes negligible in combination with an ID batch size of 100 or more. In comparison, the ID batch size has a significantly higher impact on the throughput. Using a size of

1, we needed on average 2,098.11 ns to transfer a single ID with feedback frequency 1, and 665.17 ns with a frequency of 1,000. Yet, we can drastically reduce the transfer time by increasing the ID batch size. For example, our default configuration of an ID batch size of 10,000 and a feedback frequency of 10 reduces the transfer time to 37.76 ns. Yet, as shown in Figure 6.6, after an ID batch size of 1,000, a further increase in size only marginally improves performance. Hence, we refrained from evaluating greater batch sizes although technically we could allocate up to 2^{36} bytes of memory for the enclave [66].

To allow for better interpretation of these results, the dashed gray line in Figure 6.6 indicates the trampoline call frequency, which we previously determined to be 86.53 ns. All configurations achieving transfer times below this frequency are able to transfer the IDs to the analyzer faster than the target creates new IDs. With our prototype, we stay above this frequency for ID batch sizes below 50. Yet, all greater ID batch sizes transfer the IDs faster than the trampoline call frequency, independent from the feedback frequency.

Having evaluated the throughput of the target and the queue, we continued by evaluating the analyzer's throughput (❹). As we consider the analyzer's performance in the offline phase less time-critical, we focused on the evaluation of the online phase. In the online phase, the analyzer verifies the target's control flow by comparing the CFG IDs with the previously recorded CFG. Additionally, it compares the number of iterations of every loop with its iteration limit. The analyzer performs those checks for all IDs received between the start and end tag (Section 6.4.3). With our signing service, the target sends the start tag as soon as the worker thread accepts an incoming connection, and the end tag when the connection is terminated. This permits us to attest handling of a request without having to consider other tasks such as the service waiting for a new connection.

To determine the throughput of the analyzer, we measured the time required to process the IDs of 100,000 requests. For each request, the analyzer processed around 630 IDs. From these 630 IDs, 90.93% were CFG IDs, and 9.61% loop IDs. Considering both CFG IDs and loop IDs, the analyzer required on average 9.24 ns to process a single ID. This is significantly lower than the target's trampoline call frequency of 86.53 ns. In other words, the analyzer is able to process incoming IDs faster than they are produced by the target.

6.5.4 Signing Service Performance Evaluation

After evaluating the components separately, we continued with evaluating the overhead of the entire prototype using our signing service as target. To quantify the overhead of our CFA and loop verification, we first prepared a TEE running the uninstrumented service. Using this TEE, we measured the time required by the service to process an incoming request. Using this setup, we performed 100,000 valid requests to the service, which required on average 54.79 μs to process a single request.

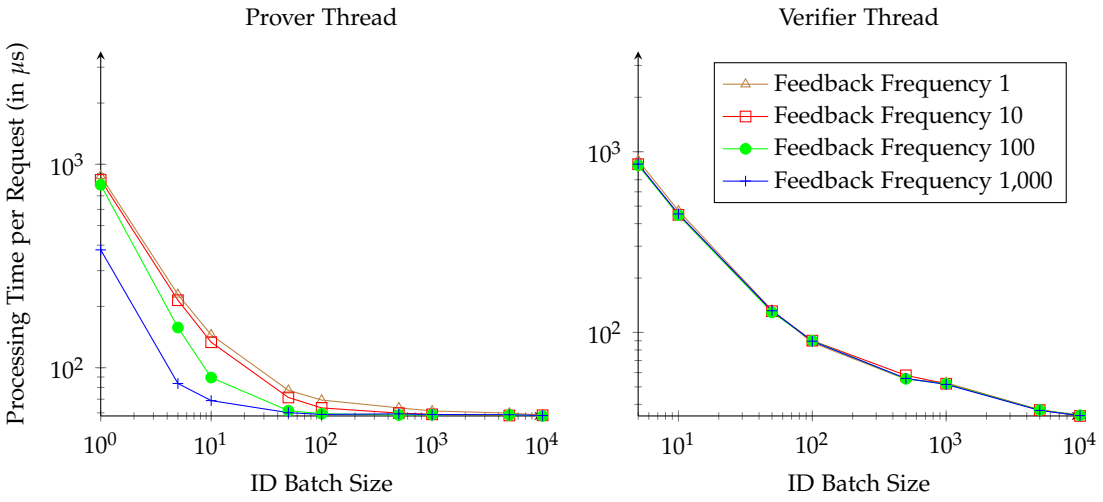

Figure 6.7: Average processing time for a single request per thread. The left graph depicts the measurements for the prover thread, and the right graph for the verifier thread.

After measuring the processing time of the uninstrumented target, we evaluated the processing time of the instrumented target. Specifically, we measured the time required by the prover thread to process the request, cache the IDs produced by the trampoline, and store the encrypted ID batches in the queue. Additionally, we determined the time taken by the verifier thread to read the encrypted ID batches from the queue, decrypt them, analyze the respective IDs, and store the result in the attestation log. Note that we are not aware of any possibility in SGX to synchronize multiple threads running in different enclaves (Section 6.4.3). Therefore, the verifier thread is required to actively check if new ID batches exist in the queue after it finishes processing the current batch. In aspiration of an improved synchronization mechanism in the future, we excluded the time the verifier thread spends actively waiting for new ID batches from our measurements.

Figure 6.7 depicts the time the prover thread requires to process a single request on the left and the time the verifier thread requires to validate the control flow on the right. To determine the impact of different ID batch sizes and feedback frequencies, we again performed the measurements for different parameter combinations. Specifically, the X-axes show the different ID batch sizes, while the Y-axes indicate the average processing time per request. Furthermore, the graphs depict the different feedback frequencies. Similar to the evaluation of the queue (Section 6.5.3), the feedback frequency has only limited impact on the processing time. In comparison, the ID batch size has a higher impact on the processing time. Using a batch size of 1, the prover thread requires between 876.84 and 379.68 μs to process a single request, depending on the feedback frequency. Yet, the graph also shows that a higher ID batch size can drastically reduce this processing time. For example, our default configuration of an ID batch size of

Table 6.2: Overheads for processing requests in the signing service for different ID batch sizes and feedback frequencies.

	Feedback Frequency			
	1	**10**	**100**	**1,000**
Batch Size 1	15.00x	14.29x	13.46x	5.93x
Batch Size 10	1.65x	1.44x	0.63x	0.26x
Batch Size 100	0.26x	0.16x	0.08x	0.08x
Batch Size 1,000	0.12x	0.08x	0.07x	0.07x
Batch Size 10,000	0.07x	0.07x	0.06x	0.06x

10,000 and a feedback frequency of 10, which we also used for the evaluation of the benchmark (Section 6.5.2), reduced the processing time to 58.36 μs. This represents a 0.07x overhead in comparison to the original response time of 54.79 μs. In Table 6.2, we show the overhead for further configurations. While we recorded a maximum overhead of 15.00x with both the ID batch size and feedback frequency set to 1, we were able to reduce the overhead to as little as 0.06x with different parameter combinations. This proves that while we recorded a considerable overhead under high CPU load, the impact on the target remains limited for traditional applications such as our signing service.

Compared to the prover thread, the verifier thread requires between 4,038.79 and 3,995.60 μs to process a single request with an ID batch size of 1. Again, the measurements show only a small difference between the different feedback frequencies. However, similar to the prover thread, we can drastically reduce the processing time per request by increasing the batch size. For example, the default configuration of an ID batch size of 10,000 and a feedback frequency of 10 reduces the processing time to 34.52 μs.

In summary, our evaluation shows that both the queue and the analyzer are able to process the IDs faster than they are collected by the trampoline. This eliminates the risk of creating a backlog of IDs to be analyzed when processing a large number of requests. Another conclusion from our evaluation is that the process of transferring IDs from the ProveTEE to the VerifyTEE can cause a relatively high overhead. However, by caching IDs in the ProveTEE and parallelizing the execution of the target and the analysis, we can limit this overhead to as little as 0.06x.

6.6 Discussion

Our design allows for the detection of control data and DOP-like attacks in cloud environments. In this section, we discuss the attacks we can avoid, and those we cannot avoid.

Missing Context-Sensitive CFA The current implementation of our prototype verifies the edges of the control flow, but does not consider its context. This prevents us from

detecting attacks that mix two different but valid control flows [25]. For example, let us assume that the target has two valid control flows: $A \rightarrow B \rightarrow C$ and $D \rightarrow B \rightarrow E$. If B contains a vulnerability, the attacker can execute the invalid control flow $A \rightarrow B \rightarrow E$. In this case, the analyzer would not be able to detect the attack, as it only considers the edges between two nodes in the CFG, but does not analyze the context of the control flow. Previous work [1] solves this limitation by storing entire valid control flows instead of only storing valid edges. Yet, the amount of possible control flows for a target drastically increases with its complexity. For example, when the target contains a loop, it would be necessary to store a separate control flow for each valid number of loop iterations. Instead, we chose to not only support very limit targets and in return to not be able to detect attacks mixing two different but valid control flows. Future work is required to determine how context-sensitive CFA can be performed on complex targets.

Detection of Non-Control Data Attacks Our current mechanism is limited to the detection of DOP-like attacks that increase the number of loop iterations in order to execute arbitrary code [63]. Therefore, we are not able to detect non-control data attacks which influence the control flow in other ways, for example by overwriting an authentication variable. Future work is required to determine how such attacks can be detected without the need for memory safety.

Integrity of the TEEs Our design relies on both the ProveTEE and the VerifyTEE to provide confidentiality and integrity protection even in the presence of a malicious administrator. However, previous research has shown that current implementations of TEEs can be suspect to attacks which violate these goals [148, 112, 29, 124, 149, 109, 21]. Such attacks will also have an effect on our design, as we would not be able to ensure the integrity of any of its components. Yet, the respective publications also discuss defense strategies, which we assume to be deployed.

Malicious HA Launching Modified TEEs Our prototype relies on the untrusted HA to launch both TEEs and to provide the shared memory holding the queue (Section 6.3.1). This enables a malicious HA to perform a Denial of Service attack by refusing to launch one or both TEEs. Unfortunately, it is difficult to prevent such attacks in general. However, while not being able to prevent them, we can use the built-in static remote attestation for the respective TEE to determine if both TEEs have been launched [73, 75]. Furthermore, the static remote attestation permits us to verify whether the TEEs have been launched without modifications.

Gadgets in Trusted Code Previous work showed that an attacker exploiting a vulnerability within an SGX enclave is able to access a variety of gadgets in trusted code such as SGX libraries [88, 14]. Our prototype is able to detect such attacks by XORing the CFG IDs of indirect branches with the offset between the current instruction pointer and the jump destination (Section 6.4.2). This safeguard allows us to detect an attacker

overwriting the jump destination to point to trusted code, as this will influence the recorded CFG ID.

Sending Malicious IDs to the VerifyTEE Another attack an adversary in control of the ProveTEE could attempt would be to compromise the VerifyTEE by sending it malicious CFG IDs. To defend against such attacks, we hardened the ID parser in the VerifyTEE and reduced it to less than 60 lines of code.

Manipulation of Cached IDs To reduce the overhead of transferring IDs from the ProveTEE to the VerifyTEE, we use ID batches to cache collected IDs. To protect against a compromised ProveTEE modifying or deleting these IDs, we use static attestation to ensure that the ProveTEE is trustworthy at launch. Hence, an attacker first needs to compromise the target to gain control over the ProveTEE and to manipulate cached IDs. Yet, before the target executes a malicious control flow, the trampoline will create at least one ID identifying the compromise, and add it to the hash chain (Section 6.4.2). When the attacker finally gains accesses to the ProveTEE, only the current value of $hash_p$ already including these IDs is accessible. To modify or delete these IDs without being detected, it would be necessary to calculate a $hash'_p$ matching the new sequence of IDs, which requires knowledge of the previous values of the non-reversible $hash_p$. To infer these previous values, a malicious administrator could attempt to collect previously exchanged ID batches, which contain previous values of $hash_p$. We protect against this attack vector by encrypting the ID batches before sending and using an irreversible KDF to calculate a new encryption key for every exchange. Hence, the attacker will not be able to decrypt previously sent ID batches even when having access to the current encryption key. This ensures the integrity of IDs cached in the ID batch even in the presence of a malicious administrator additionally gaining control over the ProveTEE.

Malicious HA Interfering with Batch Transfer The queue we use to exchange ID batches between the ProveTEE and the VerifyTEE is located in shared memory. This enables a malicious administrator to access batches stored in the queue. To prevent inspection or modification of the ID batches, we encrypt them with AES-GCM (Section 6.4.1). Yet, an attacker may also attempt to remove the ID batches from the queue before they can be read by the analyzer. In order to detect such attacks, we make use of the feedback frequency. Specifically, the feedback frequency states after how many ID batches the VerifyTEE acknowledges their receipt. If the VerifyTEE does not acknowledge the receipt of previously sent ID batches, the ProveTEE can conclude that some or all of the collected IDs are not being verified and halt execution. Similarly, the feedback frequency also helps detecting a high privileged attacker pausing or stopping the VerifyTEE. To be precise, if the VerifyTEE does not acknowledge the receipt of previously sent ID batches, the ProveTEE can conclude that the collected IDs are not being verified and halt execution.

An additional safeguard against the removal of batches from the queue are the counters we use in the ProveTEE and VerifyTEE as IV for encryption and decryption of

transmitted ID batches. To be precise, both the ProveTEE and the VerifyTEE increase their counter used as IV after each use (Section 6.3.1). Therefore, if the VerifyTEE does not receive an ID batch sent by the ProveTEE, the counters run out of sync. This means that if a batch is deleted from the queue, the VerifyTEE would attempt to decrypt the next batch with the wrong IV, detecting that a previous batch was not received.

Malicious HA Launching Multiple TEE Instances Another advantage of using counters as IV is that they enable us to detect a malicious administrator launching two instances of the VerifyTEE. In this scenario, the attacker could attempt to forward certain ID batches indicating attacks to one instance of the VerifyTEE and batches with regular IDs to another instance. Yet, such an attack would cause the counters in the ProveTEE and the VerifyTEE serving as IV for AES-GCM to run out of sync, allowing us to detect the attack. Furthermore, we are also able to detect multiple instances of the ProveTEE. Specifically, the target in the ProveTEE will be shipped without secrets such as the signature key required by our signing service. This is the normal procedure for TEE images, as the untrusted party launching the TEE has access to the image and could therefore retrieve secrets shipped with the image. Instead, secrets are only provisioned after the TEE is successfully attested. Therefore, when a malicious entity launches multiple instances of the ProveTEE, only one instance would be provisioned with the secret. This prevents the creation of multiple functional instances of the ProveTEE at the same time.

6.7 Summary

In this chapter, we investigated our third research question, *RQ3: How can we detect the exploitation of software vulnerabilities within TEEs?*. Taking one step at a time, we first discussed how we can enable a TEE to collect run-time information from an application in Section 6.2. For this purpose, we presented our contribution *C5*, an architecture that permits a monitoring process running in a TEE to collect run-time information from an application running in a different TEE. Using the example of Intel SGX, we showed how the HA can be used to launch both TEEs. This causes both TEEs to share the same address space, facilitating data exchange between them.

Based on the established architecture, we presented our contribution *C6* in Section 6.3. *C6* is a design that performs CFA in TEEs, focused on cloud environments. In our design, we instrument the application running in a TEE to regularly provide information about its control flow to a trampoline. This trampoline forwards the collected information to an analyzer hosted in a different TEE. By using TEEs, we can protect the application and the CFA mechanism from a malicious administrator. Furthermore, the separation into two TEEs allows us to protect the CFA mechanism from an attacker compromising the application. Our improved CFA mechanism not only permits the detection of control data attacks, but also of certain non-control data attacks increasing the number of loop iterations, as done for DOP.

In Section 6.4, we presented our prototype for CFA in TEEs which we implemented using Intel SGX. Our prototype reduces its performance overhead by deploying a caching mechanism for the exchange of data between the trampoline and the analyzer. To ensure that the caching mechanism does not negatively affect the prototype's security, we implemented different protection mechanisms. Example mechanisms are the deployment of a hash chain and the encryption of exchanged data with iteratively updated encryption keys.

To analyze the performance of our prototype, we provided a detailed evaluation in Microsoft Azure in Section 6.5. The deployment of a benchmark as target application shows that the additional verification can require a considerable amount of time under high CPU load. Yet, we also showed that this effect is weakened on traditional applications by deploying a signing service as target and evaluating the separate performance of the components as well as of the entire prototype. In detail, the evaluation of the signing service showed that by combining our caching mechanism with multithreading, we were able to limit the overhead in the service's response time to as little as 0.06x.

Additional to the performance evaluation, we presented a detailed security discussion (Section 6.6) in which we showed how we defend against various attacks. Furthermore, we also determined attacks that require future work for efficient prevention, namely non-control data attacks and attacks modifying the control flow, but using only valid edges.

Also note that while we implemented our prototype with Intel SGX, our design itself does not require a specific TEE. Hence, we do not impose any requirements on the cloud provider aside from the provision of a TEE. This makes our design suitable for the majority of popular cloud environments, enabling cloud users to ensure the run-time integrity of their workloads even in such environments.

CHAPTER 7

Conclusions

In this thesis, we set out to analyze and improve the security of two protection mechanisms of Trusted Execution Environments (TEEs), memory isolation and attestation. Our first set of contributions analyzed the impact of missing memory integrity on a TEE's cryptographic memory isolation on the example of AMD Secure Encrypted Virtualization (SEV). With our contributions, we showed that such missing memory integrity can not only affect the TEE's confidentiality, but also its integrity. Furthermore, we provided a framework that allows to also perform the analysis on upcoming TEEs such as Intel Trusted Domain Extensions (TDX) [69], IBM Protected Execution Facility (PEF) [64] and ARM Confidential Computing Architecture (CCA) [8]. Having discussed possible attacks, our next contribution was a countermeasure against attacks exploiting SEV's missing memory integrity by modifying the Second Level Address Translation (SLAT).

In order to further improve the security of TEEs, we also discussed approaches for dynamic attestation, permitting to detect integrity violations at run-time. Specifically, we presented an architecture that enables a monitor in a TEE to retrieve run-time information of an application hosted in a different TEE. The architecture not only allows to gather information of the application's run-time integrity, but additionally protects both monitor and application from high privileged adversaries. Based on this architecture, we presented a design that ports Control-Flow Attestation (CFA) to TEEs, permitting to detect clients exploiting software vulnerabilities within the TEE at run-time.

Having presented our contributions, we will now discuss how and in which degree they address our research questions. Finally, we conclude the thesis by identifying open research questions and possible future work.

7.1 Contributions to Research Questions

The contributions we made in this thesis were motivated by the research questions we defined in Section 1.1. In this section, we iterate through our research questions and discuss how we addressed them with our contributions.

RQ1: Which Attack Vectors Enable Attacks Against a TEE's Cryptographic Memory Isolation Without Memory Integrity? To understand the impact caused by missing memory integrity on cryptographic memory isolation, we performed a security analysis on the example of AMD SEV, which does not provide such a protection mechanism. Our analysis revealed two attacks against the TEE's confidentiality and integrity, and identified six different attack vectors. Out of these six attack vectors, we identified the modification of the SLAT by the malicious administrator as the most critical one. Another result of our analysis is a framework which allows for the exploitation of the different attack vectors[1].

RQ2: How Can We Restore Cryptographic Memory Isolation of TEEs Without Memory Integrity? Sticking to the example of SEV, we continued by analyzing different countermeasures for the previously determined attack vectors. Specifically, we discussed countermeasures that could be applied by either the Virtual Machine (VM) or SEV itself to protect against the previously described attacks. Most importantly, we proposed a small modification to the tweak mechanism applied by the AMD Secure Processor (SP) before encrypting data in the VM's memory. In detail, we suggested to incorporate the Guest Physical Address (GPA) into the tweaking mechanism to protect against a meaningful modification of the VM's memory mapping via SLAT modification.

RQ3: How Can We Detect the Exploitation of Software Vulnerabilities Within TEEs? Finally, we presented an architecture that enables a monitor located in a TEE to collect run-time information from an application located in another TEE. Based on this architecture, we proposed a design that makes use of CFA to verify the application's control flows. In comparison to previous work, we applied three modifications to our CFA mechanism, allowing us to improve previous mechanisms and to adapt them to cloud environments.

7.2 Future Work

During the investigation of our research questions, we also identified possible future work. Most importantly, the research area of run-time integrity deserves further research.

Security Analysis of AMD SEV-SNP In the first part of this thesis, we investigated the impact of missing integrity protection on cryptographic memory isolation on the example of AMD SEV. The latest version of SEV, SEV Secure Nested Paging (SEV-SNP), aims to provide integrity protection by preventing a malicious hypervisor from writing the memory of the protected VM or modifying its memory mapping [3]. Therefore, we propose an in-depth security analysis of the protection mechanisms of AMD SEV-SNP. To be precise, various techniques, such as DRAM disturbance [79] or DMA [11, 16], could make it possible for a high privileged adversary to modify the VM's memory.

[1] `https://github.com/Fraunhofer-AISEC/severed-framework`

Therefore, future work is necessary in order to verify if SEV-SNP considers all of those techniques and protects against them.

Detection of Non-Control Data Attacks Our design porting CFA to TEEs allows to detect control data attacks and some non-control data attacks. Yet, it is not able to detect all non-control data attacks. Therefore, future work is required to determine how non-control data attacks that do not modify the number of loop iterations, but instead, for example, overwrite an authentication variable, can be detected. Previous work proposes to protect against such threats by deploying memory safety technologies. However, such technologies usually incur a high performance penalty, as they require to verify all memory accesses.

Porting the Design for CFA to different TEEs Our prototype for CFA in TEEs makes use of Intel Software Guard Extensions (SGX). While our design is generic and can be applied to all TEEs, a prototype using a different TEE would have to find a method to efficiently transfer data between the ProveTEE and the VerifyTEE. A potential candidate for an implementation on a different TEE would be AMD SEV-SNP. Specifically, SEV-SNP is the first version in the SEV family that provides Virtual Machine Privilege Levels (VMPLs) [3]. VMPLs allow to separate an SEV-SNP-protected VM into up to four hardware isolated abstraction layers. The hardware ensures that memory pages assigned to a specific VMPL cannot be accessed by lower privileged VMPLs. However, VMPLs are able to access the memory of lower privileged VMPLs. This properties could be used to port our design for run-time integrity protection of TEEs to SEV-SNP. In detail, one could place the prover and the verifier functionality in different VMPLs instead of different TEEs. By placing the verifier in a higher privileged VMPL, it would be able to access the prover's memory and to retrieve collected IDs. At the same time, the verifier would be protected from an adversary gaining control over the prover running in a lower privileged VMPL. In comparison to our prototype using Intel SGX, this approach could reduce the overhead required when exchanging IDs, and therefore improve the overall performance.

Attestation of Complex Programs Another question that remains is how to establish the Control-Flow Graph (CFG) of complex applications as both static and dynamic analysis face certain limitations. Specifically, while static analysis cannot include edges that are calculated at run-time, dynamic analysis makes it difficult to determine the point at which the target has executed all valid control flows at least once. To facilitate the attestation of complex programs, future work could investigate the approach of already considering the complexity of the CFG during the target's development. This idea is based on the approach of test-driven development, in which first test cases are specified [12]. Afterwards, the software is developed in order to pass the test cases. Similarly, already considering the impact of different decisions on the CFG during

development may allow to drastically reduce the complexity of the CFG in the final software, facilitating CFA.

List of Figures

2.1 Memory Management in a Virtualized Environment 12
2.2 Receiving Data via Split Virtqueues 13
2.3 AMD SEV Overview 15
2.4 Control Flow with Intel SGX 17
2.5 Legal and Illegal Control Flows in a CFG 18

4.1 Attacking Confidentiality of a VM 32
4.2 Identifying a Secret Within the VM Via Page Tracking 39
4.3 Evaluation of Resource Identification 43
4.4 Evaluation of Secret Identification 46
4.5 Attacking Integrity of a VM 54
4.6 Packet Buffer 58
4.7 Remapping Trigger Page to Secret Page 60

6.1 Using a TEE to Verify the Run-Time Integrity of an Application 79
6.2 Design Overview 80
6.3 CFG Annotation in the Offline Phase 85
6.4 Prototype Overview 87
6.5 Exchanging IDs Between ProveTEE and VerifyTEE 88
6.6 Evaluation of ID Transfer Time 96
6.7 Evaluation of Processing Time per Thread 98

List of Tables

3.1 Overview of Software Attacks Against SEV and SEV-ES 23

4.1 Evaluation of Top Set Size 44
4.2 Evaluation of NRQ-Noise Probability and Recording Size 44
4.3 Evaluation of Secret Identification 48
4.4 Evaluation of Secret Extraction 50
4.5 Evaluation of Code Injection 62

5.1 Overview of Attack Vectors Addressed by Countermeasures 75

6.1 Evaluation of Benchmark 95
6.2 Overheads for Processing Requests in the Signing Service 99

Acronyms

ASLR Address Space Layout Randomization. 7, 71, 90, 91

CCA Confidential Computing Architecture. 6, 16, 22, 95

CFA Control-Flow Attestation. 3, 4, 7, 9, 11, 17, 19, 20, 65, 67, 69, 70, 73–75, 80, 81, 84, 87, 89–92, 95, 97, 98, 108

CFG Control-Flow Graph. 18–20, 70–73, 75, 77–81, 84, 87, 90, 91, 97, 98, 107, *Glossary:* Control-Flow Graph

CFI Control-Flow Integrity. 91

DMA Direct Memory Access. 46, 62

DOP Data-Oriented Programming. 7, 19, 20, 69, 70, 72–74, 77, 86, 90–92, *Glossary:* Data-Oriented Programming

FDE Full Disk Encryption. 22, 23, 32, 34, 35, 38–40, 55, 56, 62

GFN Guest Frame Number. 12, 24, 25, 29, 31, 32, 44–50, 52, 53, 55–59, 62, 63, *Glossary:* Guest Frame Number

GPA Guest Physical Address. 6, 12, 21, 55, 58, 59, 62, *Glossary:* Guest Physical Address

GVA Guest Virtual Address. 12, 21, *Glossary:* Guest Virtual Address

HA Host Application. 16, 17, 67, 68, 75, 81, 82, 87–89, 92, *Glossary:* Host Application

IV Initialization Vector. 76, 88

KASLR Kernel Address Space Layout Randomization. 22, 23, 45, 52

KDF Key Derivation Function. 75, 76, 80, 88

KVM Kernel-based Virtual Machine. 32

NMI Non-Maskable Interrupt. 44–46, 48–50, 52, 53, *Glossary:* Non-Maskable Interrupt

NX No Execute. 24

ORAM Oblivious RAM. 55

OS Operating System. 16, 17, 21, 22, 42, 44, 45, 52, 54, 62, 67, 68, 71, 75, 81, 89

P Present. 24

PEF Protected Execution Facility. 6, 16, 22, 95

PTE Page Table Entry. 14, 15, 24, 55, *Glossary:* Page Table Entry

R/W Read/Write. 24

RHEL RedHat Enterprise Linux. 50

RMP Reverse Map Table. 59

ROP Return-Oriented Programming. 19, 78, 90

SEV Secure Encrypted Virtualization. iii, v, vi, 2, 3, 5, 6, 8, 9, 11, 14, 15, 21–25, 34, 41, 43, 46, 49–52, 54–57, 59–63, 67, 95–98, 108

SEV-ES SEV Encrypted State. 15, 25, 46, 50, 52, 56, 59–61

SEV-SNP SEV Secure Nested Paging. 15, 59, 63, 70, 97, 98

SFN System Frame Number. 12, 24, 29, 31, 32, 44, 45, 48–50, 53, 56–58, 62, 63, *Glossary:* System Frame Number

SGX Software Guard Extensions. iv, vi, 6, 9, 11, 16, 17, 65–67, 70, 74, 75, 80, 82, 85, 92, 98

SLAT Second Level Address Translation. 12, 23, 24, 31, 32, 44, 46, 48, 49, 53, 55, 58, 59, 96, 97, *Glossary:* Second Level Address Translation

SLES SUSE Linux Enterprise Server. 50, 51

SME Secure Memory Encryption. 14, 15

SP Secure Processor. 14, 21, 57–60, 96, *Glossary:* Secure Processor

SPA System Physical Address. 12, 14, 21, 58, 59, *Glossary:* System Physical Address

TCB Trusted Computing Base. 16, *Glossary:* Trusted Computing Base

TDX Trusted Domain Extensions. 6, 8, 16, 22, 95

TEE Trusted Execution Environment. iii–vi, 2–7, 9, 11, 15–17, 19, 21, 22, 24, 61–63, 65–70, 73–75, 81, 84, 85, 87, 89–93, 95–98, 108

TLB Translation Lookaside Buffer. 17, 31

TLS Transport Layer Security. 36

vCPU virtual CPU. 53

VM Virtual Machine. iii, 1–3, 5, 6, 8, 9, 11–16, 21–26, 29–32, 34, 35, 37–62, 66, 80, 82, 96–98, 107, 108

VMPL Virtual Machine Privilege Level. 98

Glossary

Control-Flow Graph Graph representing all legitimate control flows of an application. 18, 70, 97

Data-Oriented Programming Advanced non-control data attacks that chain arbitrary gadgets while staying within the CFG. 7, 19, 69

Ecall A call to a function within an Intel SGX enclave. 16, 17

Edge Transition between nodes in a CFG. 18, 19, 71, 75, 77, 87, 90–92

Enclave Trusted part of an Intel SGX process. 16, 17, 67, 80, 85, 87

Guest Frame Number Memory page of a VM. 12, 24

Guest Physical Address Physical address used within a VM. 6, 12, 21

Guest Virtual Address Virtual address used within a VM. 12, 21

Host Application Unstrusted part of an Intel SGX process. 16, 67, 75

Hypervisor A combination of hardware and software mechanisms that manages VMs. iii, 1–3, 5, 8, 11–15, 21, 22, 24, 25, 29–32, 35, 41, 44–47, 49–52, 54–60, 62, 66, 96, 97, 108

Iteration limit Maximum number of iterations performed by a loop in the offline phase. 70–73, 75, 78, 79, 81, 84, 97

Microservice Simple service with a single functionality. 19, 71, 81

Node Uninterruptible instruction sequence in a CFG. 18, 73, 75, 87, 91

Non-Maskable Interrupt A hardware interrupt that cannot be disabled or ignored by CPU instructions. 44

Ocall A call to a function outside of an Intel SGX enclave from within the enclave. 16, 17, 81, 82

Packet Buffer Contiguous memory used within a VM to store incoming network packets. 47, 48, 50, 51, 53

Page Table Entry An entry in the page table containing the frame number and different descriptors. 14, 24

Prover Party providing information of its control flow to the verifier. 17, 19, 90

Second Level Address Translation Twofold translation from guest virtual to guest physical to system physical addresses. 12, 23, 96

Secure Processor A dedicated processor managing the encryption keys for AMD SEV. 14, 21, 96

System Frame Number Memory page of a physical system. 12, 24

System Physical Address Physical address used by the hypervisor. 12, 21

Target Application to be attested with CFA. 3, 4, 17–19, 66–75, 77–89, 91, 92, 98

Trusted Computing Base All components a TEE is required to trust. 16

Verifier Party verifying the control-flow information from the prover. 17–19, 70, 80, 81, 90, 91

Virtio An I/O virtualization framework. 12, 23, 44, 47, 54

Bibliography

[1] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik. "C-FLAT: Control-Flow Attestation for Embedded Systems Software". In: *ACM SIGSAC Conference on Computer and Communications Security*. 2016.

[2] T. Abera, R. Bahmani, F. Brasser, A. Ibrahim, A.-R. Sadeghi, and M. Schunter. "DIAT: Data Integrity Attestation for Resilient Collaboration of Autonomous Systems". In: *ISOC Network and Distributed System Security Symposium (NDSS)*. 2019.

[3] Advanced Micro Devices. *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. 2020.

[4] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual (Volumes 1-5)*. 2020.

[5] Advanced Micro Devices. *GitHub - AMDESE/AMDSEV: AMD Secure Encrypted Virtualization*. `https://github.com/AMDESE/AMDSEV`. Accessed: 2021-05-10. 2018.

[6] Amazon Web Services, Inc. *What are Microservices?* `https://aws.amazon.com/microservices/`. Accessed: 2022-05-30. 2022.

[7] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. "Innovative technology for CPU based attestation and sealing". In: *Proceedings of the Hardware and Architectural Support for Security and Privacy*. 2013.

[8] ARM Holding. *Arm Confidential Compute Architecture*. `https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture`. Accessed: 2021-10-08. 2021.

[9] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. "Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World". In: *ACM Conference on Computer and Communications Security (CCS)*. 2014.

[10] BBC. *AWS: Amazon web outage breaks vacuums and doorbells*. `https://www.bbc.co.uk/news/technology-55087054`. Accessed: 2022-05-11. 2020.

[11] M. Becher, M. Dornseif, and C. N. Klein. “FireWire: all your memory are belong to us”. In: *Proceedings of CanSecWest* (2005).

[12] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley Professional, 2003.

[13] M. Ben-Yehuda, J. Mason, J. Xenidis, O. Krieger, L. Van Doorn, J. Nakajima, A. Mallick, and E. Wahlig. “Utilizing IOMMUs for Virtualization in Linux and Xen”. In: *Ottawa Linux Symposium*. 2006.

[14] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi. “The Guard’s Dilemma: Efficient Code-Reuse Attacks Against Intel SGX”. In: *USENIX Security Symposium*. 2018.

[15] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)*. RFC 4492. 2006.

[16] A. Boileau. “Hit by a bus: Physical access attacks with Firewire”. In: *Presentation, Ruxcon* (2006).

[17] R. Boivie and P. Williams. *SecureBlue++: CPU support for secure executables*. Tech. rep. 2013.

[18] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, and A.-R. Sadeghi. “DR. SGX: Automated and adjustable side-channel protection for SGX using data location randomization”. In: *Annual Computer Security Applications Conference (ACSAC)*. 2019.

[19] A. Brown and G. Wilson. *The Architecture of Open Source Applications: Elegance, Evolution, and a Few Fearless Hacks*. Vol. 1. Lulu.com, 2011.

[20] R. Buhren, F. Hetzelt, and N. Pirnay. “On the Detectability of Control Flow Using Memory Access Patterns”. In: *Workshop on System Software for Trusted Execution (SysTEX)*. 2018.

[21] R. Buhren, H.-N. Jacob, T. Krachenfels, and J.-P. Seifert. “One Glitch to Rule Them All: Fault Injection Attacks Against AMD’s Secure Encrypted Virtualization”. In: *ACM Conference on Computer and Communications Security (CCS)*. 2021.

[22] R. Buhren, C. Werling, and J.-P. Seifert. “Insecure Until Proven Updated: Analyzing AMD SEV’s Remote Attestation”. In: *ACM Conference on Computer and Communications Security (CCS)*. 2019.

[23] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. “Control-Flow Integrity: Precision, Security, and Performance”. In: *ACM Computing Surveys* 50.1 (2017).

[24] N. Burow, X. Zhang, and M. Payer. “SoK: Shining Light on Shadow Stacks”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2019.

[25] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. “Control-Flow Bending: On the Effectiveness of Control-Flow Integrity”. In: *USENIX Security Symposium*. 2015.

[26] C. Cfir. *AMD-SEV: Platform DH key recovery via invalid curve attack: (CVE-2019-9836)*. `https://seclists.org/fulldisclosure/2019/Jun/46`. Accessed: 2021-27-10. 2019.

[27] D. Champagne and R. B. Lee. "Scalable architectural support for trusted software". In: *IEEE Symposium on High-Performance Computer Architecture (HPCA)*. 2010.

[28] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. "Non-Control-Data Attacks Are Realistic Threats". In: *USENIX Security Symposium*. 2005.

[29] Z. Chen, G. Vasilakis, K. Murdock, E. Dean, D. Oswald, and F. D. Garcia. "VoltPillager: Hardware-based fault injection attacks against Intel SGX Enclaves using the SVID voltage scaling interface". In: *USENIX Security Symposium*. 2021.

[30] Citrix Systems, Inc. *Citrix Hypervisor Security Update*. `https://support.citrix.com/article/CTX263477`. Accessed: 2021-19-10. 2019.

[31] Citrix Systems, Inc. *Citrix Hypervisor Security Update*. `https://support.citrix.com/article/CTX286756`. Accessed: 2021-19-10. 2020.

[32] T. Cloosters, M. Rodler, and L. Davi. "TeeRex: Discovery and Exploitation of Memory Corruption Vulnerabilities in SGX Enclaves". In: *USENIX Security Symposium*. 2020.

[33] M. Conti, E. Dushku, and L. V. Mancini. "RADIS: Remote attestation of distributed IoT services". In: *International Conference on Software Defined Systems (SDS)*. 2019.

[34] V. Costan and S. Devadas. *Intel SGX Explained*. `https://eprint.iacr.org/2016/086.pdf`. 2016.

[35] V. Costan, I. A. Lebedev, and S. Devadas. "Sanctum: Minimal Hardware Extensions for Strong Software Isolation". In: *USENIX Security Symposium*. 2016.

[36] M. Dalton. *[PATCH net-next 4/4] virtio-net: auto-tune mergeable rx buffer size for improved performance*. `https://lists.linuxfoundation.org/pipermail/virtualization/2013-November/025626.html`. Accessed: 2021-20-10. 2013.

[37] L. Davi, A.-R. Sadeghi, and M. Winandy. "Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-Oriented Programming Attacks". In: *ACM Workshop on Scalable Trusted Computing (STC)*. 2009.

[38] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi. "LiteHAX: Lightweight Hardware-Assisted Attestation of Program Execution". In: *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2018.

[39] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi. "LO-FAT: Low-Overhead Control Flow ATtestation in Hardware". In: *Annual Design Automation Conference (DAC)*. 2017.

[40] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246. 2008.

[41] Z.-H. Du, Z. Ying, Z. Ma, Y. Mai, P. Wang, J. Liu, and J. Fang. *Secure Encrypted Virtualization is Unsecure*. 2017. arXiv: 1712.05090 [cs.CR].

[42] J. Edge. *Finer-grained kernel address-space layout randomization*. https://lwn.net/Articles/812438/. Accessed: 2021-20-10. 2020.

[43] J. Edge. *Kernel Address Space Layout Randomization*. https://lwn.net/Articles/569635/. Accessed: 2021-20-10. 2013.

[44] Edited by Michael S. Tsirkin and Cornelia Huck. *Virtual I/O Device (VIRTIO) Version 1.1*. Latest version: https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html. Accessed: 2020-03-10. OASIS Committee Specification 01. 2019.

[45] eHealth Network. *Guidelines on Technical Specifications for Digital Green Certificates. Volume 4. European Digital Green Certificate Applications*. https://ec.europa.eu/health/sites/default/files/ehealth/docs/digital-green-certificates_v4_en.pdf. Accessed: 2022-01-02. 2021.

[46] ermetic. *State of Cloud Security 2021*. Accessed: 2022-05-11. 2021.

[47] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley. "Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution". In: *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2014.

[48] Fortanix. *Fortanix and Alibaba Cloud Partner to Launch SDKMS Runtime Encryption Key Management on Alibaba Cloud ECS to Protect Sensitive Data*. https://fortanix.com/company/pr/2018/10/fortanix-and-alibaba-cloud-partner/. Accessed: 2020-02-24. 2018.

[49] Free Software Foundation, Inc. *__atomic Builtins (Using the GNU Compiler Collection (GCC))*. https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html. Accessed: 2021-19-10. 2021.

[50] J. A. Garay and L. Huelsbergen. "Software Integrity Protection Using Timed Executable Agents". In: *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2006.

[51] T. Garfinkel and M. Rosenblum. "A Virtual Machine Introspection Based Architecture for Intrusion Detection". In: *ISOC Network and Distributed System Security Symposium (NDSS)*. 2003.

[52] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz. "Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding". In: *ISOC Network and Distributed System Security Symposium (NDSS)*. 2016.

[53] X. Ge, H. Vijayakumar, and T. Jaeger. "SPROBES: Enforcing Kernel Code Integrity on the TrustZone Architecture". In: *Workshop on Mobile Security Technologies (MoST)*. 2014.

[54] E. Göktaş, R. Gawlik, B. Kollenda, E. Athanasopoulos, G. Portokalidis, C. Giuffrida, and H. Bos. "Undermining Information Hiding (And What to do About it)". In: *USENIX Security Symposium*. 2016.

[55] O. Goldreich and R. Ostrovsky. "Software protection and simulation on oblivious RAMs". In: *Journal of the ACM (JACM)* 43.3 (1996).

[56] X. Guangrong. *[PATCH v3 00/11] KVM: x86: Track Guest Page Access*. `http://www.mail-archive.com/linux-kernel@vger.kernel.org/msg1076006.html`. Accessed: 2021-05-10. 2016.

[57] M. Haardt, M. Coleman, D. Vlasenko, and M. Kerrisk. *Linux Programmer's Manual PTRACE(2)*. 1993.

[58] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. "Lest We Remember: Cold Boot Attacks on Encryption Keys". In: *USENIX Security Symposium*. 2008.

[59] N. Hazut. *Capital One Breach: How It Could Have Been Prevented*. `https://www.securitymagazine.com/articles/90832-capital-one-breach-how-it-could-have-been-prevented`. Accessed: 2022-05-11. 2019.

[60] F. Hetzelt and R. Buhren. "Security Analysis of Encrypted Virtual Machines". In: *International Conference on Virtual Execution Environments*. 2017.

[61] F. Hetzelt, M. Radev, R. Buhren, M. Morbitzer, and J.-P. Seifert. "VIA: a Toolset for Analyzing Device Interfaces of Protected Virtual Machines". In: *Annual Computer Security Applications Conference (ACSAC)*. 2021.

[62] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. "Using innovative instructions to create trustworthy software solutions". In: *Proceedings of the Hardware and Architectural Support for Security and Privacy*. 2013.

[63] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang. "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks". In: *IEEE Symposium on Security and Privacy (S&P)*. 2016.

[64] G. D. Hunt, R. Pai, M. V. Le, H. Jamjoom, S. Bhattiprolu, R. Boivie, L. Dufour, B. Frey, M. Kapur, K. A. Goldman, et al. "Confidential computing for OpenPOWER". In: *ACM European Conference on Computer Systems (EuroSys)*. 2021.

[65] "IEEE Standard for Ethernet". In: *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)* (2018), pp. 1–5600.

[66] Intel Corporation. *Does the Intel® Software Guard Extensions (Intel SGX) SDK Provide a Way to Determine the Maximum Enclave Size?* `https://www.intel.com/content/www/us/en/support/articles/000089548/software/intel-security-products.html`. Accessed: 2023-02-23. 2023.

[67] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*. 2022.

[68] Intel Corporation. *Intel Software Guard Extensions Programming Reference*. https://www.intel.com/content/dam/develop/external/us/en/documents/329298-002-629101.pdf. Accessed: 2023-02-24. 2014.

[69] Intel Corporation. *Intel Trust Domain Extensions*. Tech. rep. 2020.

[70] Intel Corporation. *Rising to the Challenge - Data Security with Intel Confidential Computing*. https://community.intel.com/t5/Blogs/Products-and-Solutions/Security/Rising-to-the-Challenge-Data-Security-with-Intel-Confidential/post/1353141. Accessed: 2023-03-02. 2022.

[71] Intel Corporation. *SecuStack: Making the Cloud More Secure*. https://www.intel.com/content/www/us/en/customer-spotlight/stories/secustack-customer-story.html?wapkw=secunet. Accessed: 2020-02-24. 2020.

[72] IS Decisions. *Under a Cloud of Suspicion*. https://www.isdecisions.com/cloud-storage-security-issues/. Accessed: 2020-02-24. 2019.

[73] S. Johnson, V. Scarlata, C. Rozas, E. Brickell, and F. Mckeen. *Intel Software Guard Extensions: EPID Provisioning and Attestation Services*. Tech. rep. 2016.

[74] D. Kaplan. *Protecting VM Register State with SEV-ES*. Tech. rep. Advanced Micro Devices, 2017.

[75] D. Kaplan, J. Powell, and T. Woller. *AMD Memory Encryption*. Tech. rep. Advanced Micro Devices, 2016.

[76] P. Karnati. *Data-in-use protection on IBM Cloud using Intel SGX*. https://www.ibm.com/cloud/blog/data-use-protection-ibm-cloud-using-intel-sgx. Accessed: 2022-01-02. 2017.

[77] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang. "Remote Attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence". In: *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2009.

[78] C. Kil, E. C. Sezer, P. Ning, and X. Zhang. "Automated Security Debugging Using Program Structural Constraints". In: *Annual Computer Security Applications Conference (ACSAC)*. 2007.

[79] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors". In: *ACM SIGARCH Computer Architecture News* 42.3 (2014).

[80] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. "kvm: the Linux virtual machine monitor". In: *Proceedings of the Linux symposium*. 2007.

[81] T. Knauth, M. Steiner, S. Chakrabarti, L. Lei, C. Xing, and M. Vij. *Integrating Intel SGX Remote Attestation with Transport Layer Security*. Tech. rep. Intel Labs, 2018.

[82] B. Kopf. *Data-Sealing - Keystone Enclave 1.0.0 documentation*. `https://docs.keystone-enclave.org/en/latest/Keystone-Applications/Data-Sealing.html`. Accessed: 2021-11-03. 2020.

[83] N. Koutroumpouchos, C. Ntantogian, S.-A. Menesidou, K. Liang, P. Gouvas, C. Xenakis, and T. Giannetsos. "Secure Edge Computing with Lightweight Control-Flow Property-based Attestation". In: *IEEE Conference on Network Softwarization (NetSoft)*. 2019.

[84] E. Kushilevitz, S. Lu, and R. Ostrovsky. "On the (in) security of hash-based oblivious RAM and a new balancing scheme". In: *ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 2012.

[85] M. Larabel. *AMD Secure Encrypted Virtualization Is Ready To Roll With Linux 4.16*. `https://www.phoronix.com/scan.php?page=news_item&px=Linux-4.16-AMD-SEV-KVM`. Accessed: 2022-13-06. 2018.

[86] M. Larabel. *The Linux Kernel Enters 2020 At 27.8 Million Lines In Git But With Less Developers For 2019*. `https://www.phoronix.com/scan.php?page=news_item&px=Linux-Git-Stats-EOY2019`. Accessed: 2022-05-16. 2020.

[87] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. "Keystone: An Open Framework for Architecting Trusted Execution Environments". In: *ACM European Conference on Computer Systems (EuroSys)*. 2020.

[88] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang. "Hacking in Darkness: Return-oriented Programming against Secure Enclaves". In: *USENIX Security Symposium*. 2017.

[89] M. Li, Y. Zhang, and Y. Cheng. "CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel". In: *USENIX Security Symposium*. 2021.

[90] M. Li, Y. Zhang, and Z. Lin. "CROSSLINE: Breaking "Security-by-Crash" based Memory Isolation in AMD SEV". In: *ACM Conference on Computer and Communications Security (CCS)*. 2021.

[91] M. Li, Y. Zhang, Z. Lin, and Y. Solihin. "Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization". In: *USENIX Security Symposium*. 2019.

[92] S.-W. Li, J. S. Koh, and J. Nieh. "Protecting Cloud Virtual Machines from Hypervisor and Host Operating System Exploits". In: *USENIX Security Symposium*. 2019.

[93] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. "Architectural support for copy and tamper resistant software". In: *Acm Sigplan Notices* 35.11 (2000), pp. 168–177.

[94] LLVM Project. *LLVM Loop Terminology (and Canonical Forms)*. `https://llvm.org/docs/LoopTerminology.html`. Accessed: 2021-19-10. 2021.

[95] LLVM Project. *LLVM: Basic Block*. https://llvm.org/doxygen/group__LLVMCCoreValueBasicBlock.html. Accessed: 2022-31-05. 2022.

[96] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. "Flicker: An Execution Infrastructure for TCB Minimization". In: *ACM European Conference on Computer Systems (EuroSys)*. 2008.

[97] D. McGrew and J. Viega. "The Galois/counter mode of operation (GCM)". In: *submission to NIST Modes of Operation Process* (2004).

[98] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. "Innovative instructions and software model for isolated execution". In: *Proceedings of the Hardware and Architectural Support for Security and Privacy*. 2013.

[99] R. McLean. *A hacker gained access to 100 million Capital One credit card applications and accounts*. https://edition.cnn.com/2019/07/29/business/capital-one-data-breach/index.html. Accessed: 2022-05-11. 2019.

[100] A. McPherson, B. Proffitt, and R. Hale-Evans. *Estimating the Total Development Cost of Linux Foundation's Collaborative Projects*. https://www.linuxfoundation.org/tools/estimating-the-total-development-cost-of-linux-foundations-collaborative-projects/. Accessed: 2022-05-16. 2008.

[101] Microsoft. *Data Execution Prevention*. https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention. Accessed: 2022-08-06. 2022.

[102] Microsoft. *ReadProcessMemory function*. Accessed: 2021-06-11. 2021.

[103] Microsoft Security Response Center. *Coordinated disclosure of vulnerability in Azure Container Instances Service*. https://msrc-blog.microsoft.com/2021/09/08/coordinated-disclosure-of-vulnerability-in-azure-container-instances-service/. Accessed: 2022-05-11. 2021.

[104] G. Monro. "The concept of multiset". In: *Zeitschrift fur mathematische Logik und Grundlagen der Mathematik* 33 (1987), pp. 171–178.

[105] M. Morbitzer. "Scanclave: Verifying Application Runtime Integrity in Untrusted Environments". In: *International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. 2019.

[106] M. Morbitzer, M. Huber, and J. Horsch. "Extracting Secrets from Encrypted Virtual Machines". In: *ACM Conference on Data and Application Security and Privacy (CODASPY)*. 2019.

[107] M. Morbitzer, M. Huber, J. Horsch, and S. Wessel. "SEVered: Subverting AMD's Virtual Machine Encryption". In: *European Workshop on Systems Security (EuroSEC)*. 2018.

[108] M. Morbitzer, B. Kopf, and P. Zieris. "GuaranTEE: Introducing Control-Flow Attestation for Trusted Execution Environments". In: *IEEE International Conference on Cloud Computing (CLOUD)*. 2023.

[109] M. Morbitzer, S. Proskurin, M. Radev, M. Dorfhuber, and E. Quintanar Salas. "SEVerity: Code Injection Attacks against Encrypted Virtual Machines". In: *USENIX Workshop on Offensive Technologies (WOOT)*. 2021.

[110] Morning Consult and IBM Security. *Work from Home Study*. `https://filecache.mediaroom.com/mr5mr_ibmnews/186506/IBM_Security_Work_From_Home_Study.pdf`. Accessed: 2022-05-11. 2020.

[111] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. O'Reilly Media, Inc., 1996.

[112] A. Nilsson, P. N. Bideh, and J. Brorsson. "A survey of published attacks on Intel SGX". In: (2020).

[113] J. O'Connor, J.-P. Aumasson, S. Neves, and Z. Wilcox-O'Hearn. *BLAKE3 one function, fast everywhere*. `https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf`. Accessed: 2022-01-02. 2021.

[114] N. Ohfeld and S. Tzadik. *ChaosDB: How we hacked thousands of Azure customers' databases*. `https://www.wiz.io/blog/chaosdb-how-we-hacked-thousands-of-azure-customers-databases/`. Accessed: 2022-05-11. 2021.

[115] A. Oikonomopoulos, E. Athanasopoulos, H. Bos, and C. Giuffrida. "Poking Holes in Information Hiding". In: *USENIX Security Symposium*. 2016.

[116] M. Papermaster. *Initial AMD Technical Assessment of CTS Labs Research*. `https://community.amd.com/t5/amd-corporate-blog/initial-amd-technical-assessment-of-cts-labs-research/ba-p/416348`. Accessed: 2021-27-10. 2018.

[117] Petabridge, LLC. *Introduction to NBench | NBench*. `https://nbench.io/articles/index.html`. Accessed: 2022-01-02. 2020.

[118] N. L. Petroni Jr, T. Fraser, J. Molina, and W. A. Arbaugh. "Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor". In: *USENIX Security Symposium*. 2004.

[119] J. Pincus and B. Baker. "Beyond stack smashing: Recent advances in exploiting buffer overruns". In: *IEEE Security & Privacy* (2004).

[120] S. Pinto and N. Santos. "Demystifying arm trustzone: A comprehensive survey". In: *ACM Computing Surveys (CSUR)* (2019).

[121] N. Porter, G. Golan, and S. Lugani. *Introducing Google Cloud Confidential Computing with Confidential VMs*. `https://cloud.google.com/blog/products/identity-security/introducing-google-cloud-confidential-computing-with-confidential-vms`. Accessed: 2020-02-24. 2020.

[122] S. Proskurin, T. Lengyel, M. Momeu, C. Eckert, and A. Zarras. "Hiding in the Shadows: Empowering ARM for Stealthy Virtual Machine Introspection". In: *Annual Computer Security Applications Conference (ACSAC)*. 2018.

[123] S. Proskurin, M. Momeu, S. Ghavamnia, V. P. Kemerlis, and M. Polychronakis. "xMP: Selective Memory Protection for Kernel and User Space". In: *IEEE Symposium on Security and Privacy (S&P)*. 2020.

[124] M. Radev and M. Morbitzer. "Exploiting Interfaces of Secure Encrypted Virtual Machines". In: *Reversing and Offensive-oriented Trends Symposium (ROOTS)*. 2020.

[125] A. Rein. "Drive: Dynamic Runtime Integrity Verification and Evaluation". In: *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2017.

[126] M. Russinovich. *Introducing Azure confidential computing*. `https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/`. Accessed: 2020-02-24. 2017.

[127] V. Scarlata, S. Johnson, J. Beaney, and P. Zmijewsk. *Supporting Third Party Attestation for Intel SGX with Intel Data Center Attestation Primitive*. Tech. rep. 2018.

[128] V. Scarlata, S. Johnson, J. Beaney, and P. Zmijewsk. *Supporting Third Party Attestation for Intel SGX with Intel Data Center Attestation Primitive*. Tech. rep. Intel Corporation, 2018.

[129] M. Schneider, R. Jayaram Masti, S. Shinde, S. Capkun, and R. Perez. *SoK: Hardware-supported Trusted Execution Environments*. `https://arxiv.org/pdf/2205.12742`. 2022.

[130] M. Schwarz, C. Canella, L. Giner, and D. Gruss. *Store-to-Leak Forwarding:Leaking Data on Meltdown-resistant CPUs*. 2019. arXiv: `1905.05725 [cs.CR]`.

[131] Schwarz, Michael and Weiser, Samuel and Gruss, Daniel. "Practical enclave malware with Intel SGX". In: *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2019.

[132] H. Shacham. "The Geometry of Innocent Flesh on the Bone:Return-into-libc without Function Calls (on the x86)". In: *ACM Conference on Computer and Communications Security (CCS)*. 2007.

[133] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. "Preventing page faults from telling your secrets". In: *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2016.

[134] A. Silberschatz and P. B. Gagne Greg adnd Galvin. *Operating System Concepts*. John Wiley and Sons, Inc., 2018.

[135] B. Spengler. *Pax: The guaranteed end of arbitrary code execution*. 2003.

[136] SSD Secure Disclosure. *SSD Advisory – Oracle VirtualBox Multiple Guest to Host Escape Vulnerabilities*. `https://ssd-disclosure.com/ssd-advisory-oracle-virtualbox-multiple-guest-to-host-escape-vulnerabilities/`. Accessed: 2021-19-10. 2018.

[137] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. "Path ORAM: an extremely simple oblivious RAM protocol". In: *ACM SIGSAC Conference on Computer and Communications Security*. 2013.

[138] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas. "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing". In: *ACM International Conference on Supercomputing (ICS)*. 2003.

[139] Z. Sun, B. Feng, L. Lu, and S. Jha. "OAT: Attesting Operation Integrity of Embedded Devices". In: *IEEE Symposium on Security and Privacy (S&P)*. 2020.

[140] The kernel development community. *Kernel Self-Protection*. `https://www.kernel.org/doc/html/latest/_sources/security/self-protection.rst.txt`. Accessed: 2021-20-10. 2020.

[141] F. Toffalini, E. Losiouk, A. Biondo, J. Zhou, and M. Conti. "ScaRR: Scalable Runtime Remote Attestation for Complex Systems". In: *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 2019.

[142] F. Toffalini, M. Payer, J. Zhou, and L. Cavallaro. "Designing a Provenance Analysis for SGX Enclaves". In: *Annual Computer Security Applications Conference (ACSAC)*. 2022.

[143] VMware, Inc. *VMSA-2017-0018.1*. `https://www.vmware.com/security/advisories/VMSA-2017-0018.html`. Accessed: 2021-19-10. 2017.

[144] K. Wehrle, F. Pahlke, H. Ritter, D. Muller, and M. Bechler. *Linux Network Architecture*. Prentice-Hall, Inc., 2004.

[145] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A.-R. Sadeghi. "TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V". In: *ISOC Network and Distributed System Security Symposium (NDSS)*. 2019.

[146] O. Weisse, V. Bertacco, and T. Austin. "Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves". In: *ACM SIGARCH Computer Architecture News*. 2017.

[147] J. Werner, J. Mason, M. Antonakakis, M. Polychronakis, and F. Monrose. "The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves". In: *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2019.

[148] L. Wilke, J. Wichelmann, M. Morbitzer, and T. Eisenbarth. "SEVurity: No Security Without Integrity - Breaking Integrity-Free Memory Encryption with Minimal Assumptions". In: *IEEE Symposium on Security and Privacy (S&P)*. 2020.

[149] L. Wilke, J. Wichelmann, F. Sieck, and T. Eisenbarth. "undeSErVed trust: Exploiting Permutation-Agnostic Remote Attestation". In: *USENIX Workshop on Offensive Technologies (WOOT)*. 2021.

[150] Y. Wu, Y. Liu, R. Liu, H. Chen, B. Zang, and H. Guan. "Comprehensive VM Protection against Untrusted Hypervisor through Retrofitted AMD Memory Encryption". In: *IEEE Symposium on High-Performance Computer Architecture (HPCA)*. 2018.

[151] Xen-devel mailing list. *Alternate* `p2m` *design specification*. `https://lists.xenproject.org/archives/html/xen-devel/2015-06/msg01319.html`. Accessed: 2021-20-10. 2015.

[152] Yonhap News Agency. *S. Korea probes cyberattack on digital currency exchange*. `https://en.yna.co.kr/view/AEN20170703010400320`. Accessed: 2022-05-11. 2021.

[153] S. Zeitouni, G. Dessouky, O. Arias, D. Sullivan, A. Ibrahim, Y. Jin, and A.-R. Sadeghi. "ATRIUM: Runtime attestation resilient under memory attacks". In: *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2017.

[154] F. Zhang, J. Chen, H. Chen, and B. Zang. "Cloudvisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization". In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2011.

[155] X. Zhang, L. Van Doorn, T. Jaeger, R. Perez, and R. Sailer. "Secure Coprocessor-based Intrusion Detection". In: *ACM SIGOPS European Workshop*. 2002.

[156] Y. Zhang, X. Liu, C. Sun, D. Zeng, G. Tan, X. Kan, and S. Ma. "ReCFA: Resilient Control-Flow Attestation". In: *Annual Computer Security Applications Conference (ACSAC)*. 2021.

[157] P. Zieris and J. Horsch. "A Leak-Resilient Dual Stack Scheme for Backward-Edge Control-Flow Integrity". In: *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2018.