Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie

DZIEDZINA: Nauki Inżynieryjno-Techniczne

DYSCYPLINA: Informatyka Techniczna i Telekomunikacja

ROZPRAWA DOKTORSKA

Bezpieczeństwo wirtualnych wdrożeń z wykorzystaniem sprzętowego wsparcia platformy

Autor: Michał Kucab

Promotor rozprawy: Dr hab. inż. Piotr Chołda, prof. AGH
Promotor pomocniczy: Dr hab. inż. Piotr Boryło, prof. AGH

Praca wykonana:
Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
Wydział Informatyki, Elektroniki i Telekomunikacji
Instytut Telekomunikacji

Kraków, 2023

AGH University of Krakow

FIELD OF SCIENCE: Engineering and Technology

SCIENTIFIC DISCIPLINE: Information and Communication Technology

DOCTORAL THESIS

Security of virtual deployments with enhanced platform awareness capabilities

Author: Michał Kucab

First supervisor: Dr hab. inż. Piotr Chołda, prof. AGH
Assisting supervisor: Dr hab. inż. Piotr Boryło, prof. AGH

Completed in:
AGH University of Krakow
Faculty of Computer Science, Electronics and Telecommunications
Institute of Telecommunications

Krakow, 2023

I dedicate my dissertation work to my family and friends.

Acknowledgements

I would like to express my heartfelt gratitude to Dr hab. inż. Piotr Chołda, Prof. AGH, for his guidance, time, and careful supervision throughout this dissertation. His expertise and support have played a crucial role in shaping the final form of this research. I am truly thankful for his commitment to setting high standards, evaluating my work, and providing continuous support.

I also want to extend my sincere thanks to Dr hab. inż. Piotr Boryło, Prof. AGH, for his invaluable assistance and detailed feedback that significantly improved both the quality of the research and the dissertation itself. I greatly appreciate his dedication to excellence and constant pursuit of perfection throughout this doctoral journey.

I am deeply grateful to both Dr Chołda and Dr Boryło for their time, dedication, and invaluable contributions. Their involvement has been absolutely essential in shaping the overall results of this research.

I would like to express my appreciation for the support provided by Akamai Technologies. Their assistance in effectively managing my time, accessing necessary hardware and tools, and their unique insights in the problem space were instrumental in the successful development of this work, enabling me to achieve meaningful results.

Streszczenie

Szybki rozwój technologii zmienił nasze codzienne nawyki, umożliwiając zdalne zaspokajanie różnych potrzeb. Trend ten spowodował wzrost potrzeb na zdalne usługi dostępne przez Internet. Wirtualizacja centrów danych odegrała tu kluczową rolę, zapewniając elastyczność, szybką skalowalność, zdalne zarządzanie i szybsze wprowadzanie na rynek nowych usług. Zastąpienie fizycznych infrastruktur (tzw. *bare metal*) wirtualnymi odpowiednikami napotkało jednak na trudności związane z tym, że wirtualizacja nie oferuje porównywalnego poziomu bezpieczeństwa. Serwery fizyczne są wyposażone w sprzętowe moduły bezpieczeństwa. Stanowią one fundament bezpieczeństwa, który jest trudny do zapewnienia w maszynach wirtualnych. W rezultacie wiele wrażliwych usług wciąż korzysta z fizycznych infrastruktur. Jest to widoczne zwłaszcza wśród firm zajmujących się przetwarzaniem wrażliwych danych.

W odpowiedzi na rozwijający się trend wirtualizacji, skupiono się na procesie zdalnej atestacji maszyn wirtualnych (VM), zakładając, że działają one w niezaufanej i będącej poza kontrolą infrastrukturze chmurowej.

Zaprezentowano rozwiązanie **SGX+CET**, które umożliwia ustanowienie łańcucha zaufania do komponentu sprzętowego w środowisku chmurowym. Rozwiązanie przedstawione w ramach pracy doktorskiej opiera się na możliwościach nowoczesnych procesorów i nie wymaga dodatkowych komponentów sprzętowych. Umożliwia ono weryfikację integralności systemu plików i środowiska wykonawczego. Istniejące rozwiązania zdalnej atestacji dla infrastruktury chmurowej nie zapewniają równocześnie atestacji statycznej i dynamicznej. Pokrywają one albo część statyczną, albo dynamiczną. Wypełniono tę lukę, proponując komplementarny proces atestacji oparty na technologii sprzętowej. Pokazano, w jaki sposób ochrona wspomagana sprzętowo może zwiększyć bezpieczeństwo wirtualnej infrastruktury przy minimalnych wymaganych zmianach.

Dodatkowo, oceniono wydajność prototypu, mierząc narzut wirtualizacji, oraz

możliwości skalowania rozwiązania. Przyjęto, że małe pliki konfiguracyjne, pliki binarne i pliki wykonywalne są najbardziej krytyczne. Zaprezentowane wyniki pokazały, że rozwiązanie **SGX+CET** może zweryfikować istotne składniki systemu plików przy minimalnym wpływie na czas uruchamiania, jak również podczas działalności operacyjnej. Ponadto, czas przetwarzania może być nawet skrócony, gdy mechanizm ten jest stosowany jako domyślna metoda ochrony przed atakami związanymi z kontrolą przepływu.

Przeanalizowano również kluczowe aspekty bezpieczeństwa i pokazano, że przy odpowiednich środkach zaradczych zaproponowany model może zapewnić wymagany poziom bezpieczeństwa. Dzięki zasotosowaniu przedstawionej propozycji, wirtualna maszyna może tworzyć zaufaną infrastruktrę obliczeniowej do przetwarzania wrażliwych danych.

Podsumowując, zaproponowane rozwiązanie **SGX+CET** oferuje praktyczne i ekonomiczne podejście do zapewnienia integralności w istniejących niezaufanych infrastrukturach chmurowych. Poprzez dostarczenie dodatkowego poziomu bezpieczeństwa bez konieczności znaczących modyfikacji aplikacji lub infrastruktury, adresuje kluczowe wyzwania związane z bezpieczeństwem środowisk opartych na chmurze obliceniowej.

Abstract

The rapid evolution of technology has transformed our daily habits, allowing us to fulfill various needs remotely. This shift has resulted in an increased demand for remote services accessible over the Internet. Datacenter virtualization has played a crucial role in enabling this trend by providing flexibility, rapid scalability, remote management, and faster time-to-market. However, the replacement of bare metal deployments with virtual equivalents has been hindered by virtualization's inability to provide a comparable level of security. Physical hosts equipped with hardware security modules offer essential data security guarantees that are challenging to replicate in virtual machines. Consequently, many security-sensitive services still rely on bare metal deployments, especially among industry leaders who process sensitive data.

With response to the emerging virtualization trend, we focus on a Virtual Machine (VM) remote attestation process assuming that it is running in an uncontrolled and untrusted cloud infrastructure.

We present **SGX+CET** solution that is able to establish a chain-of-trust to a hardware component in a cloud environment. Our solution is based on built-in modern CPU capabilities and it does not need any additional hardware components to detect system modifications. Our solution enables integrity verification of a filesystem and runtime environment. Existing remote attestation solutions for cloud infrastructure do not cover static and runtime attestation as a whole. They evaluate either static or runtime part, not considering the rest. We address this gap proposing complementary attestation process based on hardware technology. We show how hardware-assisted protection can enhance virtual deployment security with minimal tradeoff.

Additionally, we evaluate the performance impact of the prototype, virtualization overhead for a real-life scenario and scaling capabilities of the solution. Here, we assume that small configuration files, binaries, and executables are the most critical. The results show that our **SGX+CET** solution can verify

important filesystem components with a minimum impact on a startup time and during operations. Moreover, a processing time can be even reduced when this mechanism is used as a default protection method against control-flow related attacks.

To enhance our solution, we analyze key security aspects imposed on remote attestation systems. We show that with necessary countermeasures, the proposed model can ensure the required level of security. This way, the whole proposal allows for making VM a part of a trusted compute resource pool.

In conclusion, our **SGX+CET** solution offers a practical and cost-effective approach to ensure integrity in existing untrusted cloud infrastructures. By providing an additional layer of security without requiring significant modifications to applications or infrastructure, it addresses crucial security challenges and holds great promise for the future of secure cloud-based environments.

Contents

Contents vii

List of Figures ix

List of Tables xi

Abbreviations xiii

Dictionary xv

1 Introduction 1
- 1.1 Thesis . . . 3
- 1.2 Structure of the dissertation . . . 5
- 1.3 Dissertation results . . . 6

2 Background 7
- 2.1 Integrity . . . 9
- 2.2 Intel SGX . . . 14
- 2.3 Intel CET . . . 16
- 2.4 Remote attestation . . . 18
- 2.5 Challenges for remote attestation in a virtualized infrastructure. . . 20

3 Related Work 23
- 3.1 Static attestation . . . 23
- 3.2 Runtime attestation . . . 26
- 3.3 Security of technologies used in our proposal . . . 30
  - 3.3.1 SGX security . . . 31
  - 3.3.2 CET security . . . 32

3.4 Summary 34

4 Static attestation **37**
4.1 Remote attestation with virtual SGX 37
4.1.1 Remote attestation of enclaves 39
4.1.2 Remote attestation of virtual machines 39
4.1.3 Proposed model 40
4.1.4 Communication details of the proposed model and our contribution in this area 40
4.1.5 Security properties 43
4.2 Comparison of virtual TPM and SGX based remote attestation . . 44
4.3 Implementation details for the attestation client and the enclave . 47
4.4 Implementation details for the attestation server 51
4.5 Evaluation of the proposed system 53
4.5.1 Performance evaluation of the attestation server 54
4.5.2 Detailed comparison 56
4.5.3 Overhead added by the execution environment 60
4.5.4 Potential improvements for real-life implementation 62
4.6 Security considerations 63

5 Runtime attestation **67**
5.1 Evaluation of the Control Flow Integrity Enforcement 68
5.1.1 Methodology of the tests 68
5.1.2 Results 70
5.2 Solution details 72
5.3 Static and runtime attestation through the entire VM lifecycle . . 73
5.3.1 Static host filesystem integrity 75
5.3.2 VM static integrity at a boot time 77
5.3.3 VM periodical static integrity 77
5.3.4 VM runtime integrity 77
5.3.5 Host runtime integrity 77
5.4 Evaluation of our proposal 77
5.4.1 Methodology of the tests 79
5.4.2 Results 80

6 Conclusions **87**

Bibliography **90**

Index **107**

List of Figures

1.1 Simplified communication model with our proposal for VM attestation. 4

2.1 Vehicle-to-Vehicle communication for self-driving cars 9
2.2 Static and runtime integrity 10
2.3 Sample control flow graph 13
2.4 Forward and backward edges. 13
2.5 SGX concept 15
2.6 Intel CET availability 16
2.7 The shadow stack concept 17
2.8 The indirect branch tracking example 18
2.9 vTPM architecture in a multi-cloud environment 21
2.10 Proposed, simplified architecture for VM static and runtime integrity verification. 22

4.1 Comparison of vTPM and vSGX architecture in a multi-cloud environment 38
4.2 Communication diagram for the proposed approach 41
4.3 Trusted and untrusted functions separation by the Enclave Definition Language 48
4.4 The bridge between the enclave and the attestation client. 49
4.5 Attestation server algorithm 52
4.6 Dependency between the number of attestations and a processing time 55
4.7 Processing time for 50 files, avg size 1 kB 57
4.8 Processing time for 50 files, avg size 1 MB 58
4.9 Processing time for 50 files, avg size 1 GB 59

4.10 Dependency between filesize and processing time for 50 files for Virtual SGX 61

5.1 Processing time overhead 71
5.2 Runtime attestation extension for prover and verifier 72
5.3 Integrity checks through VM lifecycle 73
5.4 **SGX+CET** attestation for virtual machines 75
5.5 Enhanced static host integrity validation 76
5.6 Test scenarios architecture 79
5.7 Processing time for the three workload sizes and test programs under four test scenarios 81
5.8 CET overhead for a bare metal and a virtual machine 83
5.9 Avgerage CET and virtualization overhead 84
5.10 Virtualization overhead 85

List of Tables

3.1 Comparison of solutions providing system attestation 35

4.1 Comparison of TPM vs. SGX based remote attestation for VMs that we propose 44
4.2 Lines of code for the attestation client building blocks, rounded. . 47
4.3 Processing time for a singe file (avg values) 62

5.1 Test programs execution details with median processing times for a bare metal. 70
5.2 Workload size for test programs 80

Abbreviations

ASLR	Address Space Layout Randomization
AS	Attestation Server
CAT	Cache Allocation Technology
CET	Control-Flow Enforcement Technology
CFG	Control-Flow Graph
CFI	Control-Flow Integrity
CONFIRM	Control-Flow Integrity Relevance Metrics
CRTM	Core Root of Trust for Measurement
DEP	Data Execution Prevention
EDL	Enclave Definition Language
EPA	Enhanced Platform Awareness
EPC	Enclave Page Cache
EPID	Enhanced Privacy ID
HSM	Hardware Security Module
IaaS	Infrastructure-as-a-Service
IAS	Intel Attestation Service
IBT	Indirect Branch Tracking
IMA	Integrity Measurement Architecture
JOP	Jump-Oriented Programming
JVM	Java Virtual Machine
LVI	Load Value Injection
MaaS	Metal-as-a-Service
MBA	Model-Based behavioral Attestation
MEC	Multi-Access Edge Computing
OS	Operating System
PaaS	Platform-as-a-Service
PCR	Platform Configuration Registers

RAC	Runtime Attestation Control
RAM	Runtime Attestation Module
ROP	Return-Oriented Programming
RPK	Root Provisioning Key
RSK	Root Sealing Key
SGX	Software Guard Extensions
TCB	Trusted Computing Base
TCG	Trusted Computing Group
TCS	Trusted Computing System
TEE	Trusted Execution Environment
TOCTOU	Time of Check Time of Use
TPM	Trusted Platform Module
UCON	Usage Control
V2V	Vehicle-to-Vehicle
VM	Virtual Machine
VNF	Virtual Network Function
vTPM	Virtual Trusted Platform Module

Dictionary

Below, we provide a list of terms used in this dissertation, along with their definitions or explanations. This is done to eliminate any ambiguity or confusion that may arise from terms that have different meanings depending on the context.

Application A piece of software that provides a service to end users. It is fully owned and controlled by the application vendor. As an example, an application vendor can fully manage a web server deployed in the IaaS model, but does not have a control over the underlying components.

Bare Metal A physical server dedicated exclusively to one tenant.

Chain-of-Trust In computer security context, a chain-of-trust creates a relationship between root-of-trust and the end entity. It is established by validating every component on its path.

Cloud Computing On-demand remote server resources that can be used to process and store data like on a physical host. Cloud computing is typically offered as a service, with three main models: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS).

Cloud Software Stack It is a collection of all components that build cloud computing service. As an example, for the IaaS model, it contains software that creates network, storage, server, and virtualization. For the PaaS model, an operating system and a middleware runtime are also provided.

enclave SGX enclave is a secure region of memory that is isolated from the rest of the system and protected by hardware-based encryption and access control mechanisms. The content of the enclave is encrypted and can only be accessed by the application that created it, using cryptographic keys that are generated and managed within the enclave itself. Enclaves are

designed to provide a trusted execution environment for applications that process sensitive data.

Enhanced Platform Awareness Host platform hardware capabilities, that can be exposed to a virtual machine (VM) to enable some functionality or eliminate the impact from noisy neighbours. When a virtualization is used, a virtual machine does not use a hardware directly. Moreover, a hardware is often shared with other tenants running on the same host. This impacts performance and security. With EPA in place, a hypervisor can exclusively expose required hardware directly to a VM. Then a VM can achieve near native performance and similar to bare metal security.

Hardware Security Module A physical computing device that is used to generate, store, and manage cryptographic keys and perform cryptographic operations. From a functional point of view, this hardware device is similar to Trusted Platform Module. Technically, this is a more powerful, standalone and removable device that can be accessed over the network. Because of that, it can be attached to any physical hosts inside a datacenter. Many cloud service providers offer HSM as a service, which allows users to access and use HSM in the cloud without having to manage the physical devices themselves.

Host *(noun)* A physical server connected to the Internet. It can be virtualized and offer virtual resources for guests.

Hybrid Cloud A model, where computing resources can be composed of on-premise, private and public cloud services.

Hyperscaler A company that provides cloud computing services on a massive scale, with a global presence.

Infrastructure-as-a-Service (IaaS) One of the cloud computing provisioning models that delivers core cloud capabilities, i.e. network, storage, and compute to end users. Those components then can be used to build required functionality.

Integrity System integrity refers to the quality or state of a computer system or network being secure, reliable, and consistent. It means that the system is functioning as intended, without any unauthorized changes or corruption of data.

Metal-as-a-Service A service that provides automation for bare metal provisioning, transitioning physical servers into flexible and manageable computing resources.

Multi-Cloud A cloud computing strategy that involves services from multiple cloud providers to meet different business needs. In a multi-cloud environment, an organization may use different cloud services such as Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS) from different providers, based on their specific requirements.

Platform Configuration Registers (PCR) A dedicated memory in a TPM module, used to store measurements of various system components and software, such as the BIOS, bootloader and operating system. A TPM provides a secure way to store and manage PCRs, ensuring that they cannot be tampered with or modified by unauthorized parties.

Platform-as-a-Service (PaaS) A cloud computing model in which a cloud provider offers a platform to develop, run, and manage applications without the complexity of building and maintaining the infrastructure typically required to host and run these applications. PaaS, in addition to the IaaS model, provides a platform for software creation. With that, a developer does not have to worry about the underlying infrastructure and the operating system and can focus on building the software.

Private Cloud A cloud computing model, where the infrastructure is used exclusively by a single organization and often owned by the same organization.

Public Cloud A cloud computing model, where the infrastructure and services are managed by a third-party provider. They are shared using public Internet with multiple organizations.

Root-of-Trust Hardware, software of firmware component that is inherently trusted and must be secure by design. It is a foundation for any security process in the system.

Server A hardware or software that provides a service to another device.

Service Quality The level of excellence or satisfaction experienced by end users in their interactions with a service provider.

Service A software functionality that clients can use.

Software-as-a-Service (SaaS) Cloud computing provisioning model, where software applications are hosted in a cloud. With this model, users don't need to install software on their local devices. Instead, they can access the software through a web browser or mobile application.

Trusted Computing Base A part of a computer system (hardware, firmware, or software components) that is responsible for enforcing the security policy of the system. It is the minimal set of components within a computing system that must be trusted to ensure that the system operates securely and reliably.

Trusted Computing Group An organization that defines open and vendor-neutral standards for trusted computing platforms, that is, they proposed a specification for TPM that was later standardized.

Trusted Computing A technology developed to improve the security of computing systems by establishing a trusted environment that is isolated from potential threats. It is based on a set of hardware and software technologies that work together to ensure the integrity and confidentiality of computing systems. When referring to a Trusted Computing, we consider a system that always behaves in expected way. This behavior is enforced by hardware and software that usually use a unique immutable encryption key as an anchor to build a chain-of-trust with other software components.

Trusted Execution Environment A secure area of a device that is isolated from the rest of the system and provides a trusted and secure environment for processing sensitive code and data.

Trusted Platform Module A physical chip on a motherboard that serves two major purposes: a safe memory and the encryption engine. A memory can be used to securely store sensitive information, like passwords, encryption keys, certificates or system measurements. The encryption engine is used for secure computations like key generation, hash calculation or random number generation. With these capabilities, TPM can be used to enable platform authentication and attestation necessary for secure computing.

Virtual Network Function An application that can perform network functions such as routing or caching, usually deployed as a virtual machine. VNFs were often the first step in the transition from bare metal to network virtualization, afterwards cloud-native and containerized applications were introduced.

Virtualization A process, in which software emulates the hardware functionality necessary to run another (virtual) system. Such a system is abstracted away from real hardware and software components, and instead it uses their virtual equivalents. In this thesis, by "virtualization", we mean virtualization of the server, unless specified.

1 Introduction

Over the last two decades, our everyday habits have changed a lot. Currently, we can deal with a lot of human needs remotely, e.g. we can do shopping, study, work and even visit a doctor. Therefore, the demand for remote *services* available over the Internet has increased significantly. This trend would not be possible without datacenter *virtualization* that enabled flexibility, rapid service scaling, remote management, much faster time-to-market and more other benefits. Virtualization methods have also evolved, allowing more deployment options. Nowadays, depending on the service requirements, one can choose between a virtual machine or a lightweight container for their application. We could expect that the *bare metal* deployment method will finally be replaced by its virtual equivalent to take advantage of what virtualization brings. The process cannot proceed because virtual deployments do not provide a security level comparable to what bare metal can offer [50, 88]. This is because a physical *host* can be equipped with *hardware security modules* that are required to guarantee data security. Those modules cannot be easily virtualized and hence virtual machines do not offer security fundamentals, i.e. basic principles and practices that help to ensure the confidentiality, *integrity*, and availability of data. Because of that, many services that leverage security modules for data confidentiality or integrity still use bare metal. This is visible in the industry among big companies that set directions of development in technology. They offer large-scale services and cannot risk their brand reputation when a security incident occurs.

Virtualization has created a basis for *cloud computing*. Nowadays, it is widely used in many industry areas. The biggest game changers in this industry are cloud providers and telecommunication companies. Cloud providers provide large core deployments in urbanized areas with high population density. They also offer regional deployments where the proximity to end users matters. Telecommunication companies connect mobile devices to the edge of the Internet.

They offer the closest possible location for the edge cloud computing infrastructure.

The motivation for this research originated from the fact that the security aspects of virtual deployments are not addressed properly with ongoing datacenter virtualization trends. Unfortunately, the existing security measures offered by cloud providers are not sufficient. Therefore, many companies still maintain a large number of physical *servers* around the world to provide services. Several hundred thousand servers is owned and managed by Akamai Technologies to provide secure platform accelerating and protecting Internet traffic. Those servers are used by almost any industry including banks, e-commerce or automotive. If the cloud offered a higher level of security, a company could also leverage cloud infrastructure to deploy services in a secure manner on a large scale.

With the huge demand for latency- and throughput-sensitive applications, such as online gaming, virtual and augmented reality, financial trading systems or voice and video conferencing, there is a strong need to deploy secure services at the edge of the Internet together with regional and core deployments offered by *hyperscalers*. In addition, those services should be flexible enough to handle unpredictable traffic patterns. This implies a shift towards virtualization and orchestration, as well as it requires a careful design of the new infrastructure replacing existing bare metal deployments. It is not clear for the industry what the next generation Multi-Access Edge Computing (MEC) platform or Telco Edge will look like. Operators and ISPs strive to analyze existing solutions and choose the one that can be seamlessly adapted to their existing infrastructure. They try to select an efficient solution that provides the required functionality, but they have not considered security at the required level yet.

We focused on the integrity, as one of the most critical pillars for the *application* security. With the increasing number of more sophisticated attacks that target integrity, a comprehensive approach to integrity protection is required. By a comprehensive approach we mean *static integrity* verification for files supplemented by *runtime integrity* enforcement. Those aspects should be addressed to ensure proper and trusted application behavior. Moreover, we notice that the complex architecture of any solution is often an obstacle to a successful deployment in an already complex cloud model. In our solution, in comparison to other models, we avoid complexities. This eliminates challenges related to the integration with the existing components. Furthermore, our model is easy to scale and maintain. Another motivation to keep our model less complex is the ownership of the solution. It is often shared between a cloud provider and the application vendor, which directly implies a responsibility for proper application behavior and trust. In such environment, an application vendor cannot guarantee on their application behaviour and may not decide to use available security capabilities. From the aforementioned reasons, such solutions may not gain traction from the industry.

We address this by proposing a solution that involves only components fully owned and controlled by the application vendor. Our solution leverages *Enhanced Platform Awareness (EPA)* capabilities, i.e. hardware features of the host platform it is running on, to ensure the highest possible security level. We decided to use existing hardware technologies, because it is in the interest of the company to apply this solution in production in a reasonable time frame. Intel is the only processor vendor who provides a technology that we could use in our research. We propose the enhanced remote attestation model, as shown in Fig 1.1. The existing solution in this area is grayed out, and our proposal is a second step in this model. A diagram describes a communication between system components. At this stage, we introduce our model for VM attestation, which is detailed in the following chapters.

Our method is based on Intel's Software Guard Extensions (SGX), where the hardware SGX is exposed by a hypervisor to a guest VM. Runtime integrity is ensured by the proposed runtime attestation module based on hardware technology (built into the processor), called Control-Flow Enforcement Technology (CET). Our proposal does not require changes to the cloud infrastructure. Instead, with smart selection of hardware and *cloud software stack*, it can expose important security capabilities to virtual machines. An application vendor can then ensure the highest possible level of application integrity in *Infrastructure-as-a-Service (IaaS)* environment.

1.1 Thesis

The following is the thesis statement of this dissertation:

It is possible to assure static and runtime integrity for virtual machines that leverage Enhanced Platform Awareness capabilities in an untrusted cloud infrastructure with minimal changes to the existing software stack.

The dissertation addresses key problems related to integrity in cloud environments. These problems directly impact the virtualization trend, preventing security-sensitive applications from being virtualized. The most important are the following.

- Establishing a *chain-of-trust* to a hardware *root-of-trust* in virtual deployments is challenging. This is due to the challenge of seamlessly exposing existing hardware modules to a virtual machine within a flexible IaaS environment. We show that with a proper hypervisor and processor security capabilities, a hardware root-of-trust can be directly available inside a virtual machine enabling *trusted computing*.

- Static and runtime integrity are treated separately in the literature. Because

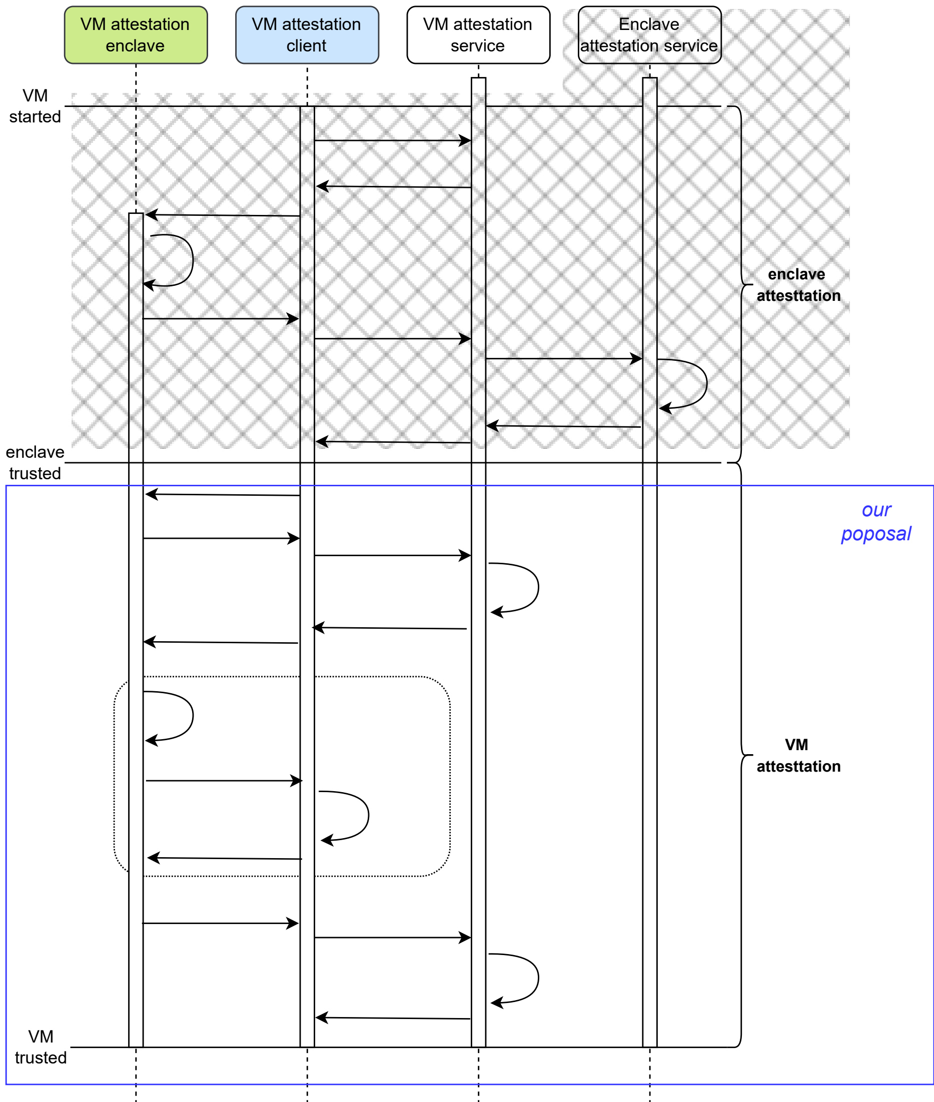


Fig. 1.1: Simplified communication model with our proposal for VM attestation.

of that, separate solutions often owned by different entities are required to ensure both integrity types for applications. This directly impacts the architecture, introducing significant complexity that is often not accepted by the application vendors. To avoid the mentioned obstacles, we propose one comprehensive solution, entirely owned by the application vendor. Such a comprehensive solution, called **SGX+CET** is a main result of this thesis. The proposed solution model is detailed in Section 4.1.3.

- Any software implementation of the security feature is considered to be less secure then its hardware equivalent. Our solution can utilize security capabilities of the hardware platform it is running on. Because of that, it is considered more reliable and difficult to compromise.

- Existing solutions for runtime integrity verification introduce an overhead that may not be acceptable and often ensure only partial protection. We prove that with dedicated hardware solution, an overhead is negligible and the runtime integrity can be enforced for the entire filesystem.

- A solution that would require major changes to the existing cloud infrastructure would certainly not be implemented. Our solution does not require them. *Minimal changes* are related to a hypervisor, that should expose hardware SGX and CET capabilities to a virtual machine. At the time of starting our research, none of those was offered by hyperscalers. At this time, SGX is available.

1.2 Structure of the dissertation

We start from a background information in Chapter 2 that provides a foundation for the study and sets the stage for the reader to understand the research problem of this thesis. Chapter 3 presents work related to static and runtime attestation problems. It describes the most significant research in this area and highlights issues that have not been fully addressed. In Chapter 4, we present our model for virtual machine remote static attestation, we evaluate performance impact when Intel's SGX-based file measurement is in place, and we show virtualization layer overhead for this process. Chapter 5 details on a mechanism for runtime attestation based on hardware control-flow enforcement. We evaluate this technology for a bare metal. Afterwards, we propose an architecture for a runtime enforcement for virtual deployments. We show how we integrate runtime attestation components with the solution proposed in Chapter 4 for static attestation to create a comprehensive integrity attestation solution for virtual deployments. Afterwards, we evaluate a performance of hardware control-flow

enforcement exposed by a hypervisor. Finally, we summarize our research in Chapter 6.

1.3 Dissertation results

As a part of the industrial PhD, we conducted research on integrity of virtual machines. Our research was consulted with Akamai Technologies, who maintains a huge number of physical servers. A company is interested in all the benefits that cloud deployment brings and would like to migrate workloads to cloud infrastructure. Before this can happen, security aspects of such deployments have to be ensured. The following papers resulted in this effort:

[72] Michał Kucab, Piotr Boryło, and Piotr Chołda. Remote Attestation and Integrity Measurements with Intel SGX for Virtual Machines. *Computers & Security*, 106:102300, 2021.

[73] Michał Kucab, Piotr Boryło, and Piotr Chołda. Performance Impact of Control Flow Enforcement Technology (CET). In *2022 25th Conference on Innovation in Clouds, Internet and Networks (ICIN)*, pages 96–100. IEEE, 2022.

[74] Michał Kucab, Piotr Boryło, and Piotr Chołda. Hardware-Assisted Static and Runtime Attestation for Cloud Deployments. *IEEE Transactions on Cloud Computing. Major revision request.*, 2023. (major revision request 14th May, 2023)

2 Background

Virtualization is a technology that has revolutionized the way datacenters operate. In the past, physical servers inside datacenters were running a separate operating system and applications. However, with virtualization, multiple virtual machines can run on a single physical server, each with its own operating system and applications. This allows for more efficient use of resources and better flexibility and scalability. Additionally, there is a growing trend towards using cloud-based virtualization solutions. Such solutions offer a scalable and cost-effective way to run virtual machines. Moreover, application providers often do not rely on a single cloud provider only, but they use a *multi-cloud* configuration. With multi-cloud approach, an application vendor can provide better service quality in terms of throughput, latency and better coverage for their services. For any company, a multi-cloud strategy with cloud-agnostic infrastructure brings the most benefits. The reason behind this is their ability to maximize cost efficiency in virtual infrastructure by transferring certain workloads to a secondary cloud that offers more affordable services. They also avoid vendor lock-in and increase redundancy for services they operate.

With all these benefits offered by the multi-cloud approach, virtualization has introduced new challenges. Nowadays, applications may not be running in dedicated trusted hardware, but rather in a multi-tenant infrastructure where the hardware is shared. This trend impacts the application's security, as noticed by the industry and described by *Trusted Computing Group (TCG)* [52]. According to the conventional approach, the application is deployed on a bare metal and can directly leverage hardware resources. As an example, for performance, the application can exclusively use a whole network card or a processor. For the security, the operating system and the application can benefit from *Trusted Platform Module (TPM)* for secure key storage or encryption engine. Similar situation, where a hardware is exclusively allocated for one service happens for *Metal-as-a-Service*

(MaaS) deployment model. This model is used when the application requires so many resources that provisioning of the entire physical server is reasonable. Hyperscalers offer bare metal or MaaS deployments, but this is not a preferred deployment method for cloud-ready services. For these reasons, the IaaS model is the natural way to deploy flexible and highly distributed multi-cloud applications.

Additionally, the conventional approach to the performance and security aspects, where a hardware can be used directly in the application, must be reevaluated, as explained by Pearson [97]. Nowadays, public cloud providers offer comparable level of security for IaaS model. Their compute hosts do not enable trusted computing, a term referring to technologies that ensure computing security through a hardware and software stack. Instead, they offer generic processing capabilities that can meet *the average customer* expectations. This means that security-sensitive applications cannot be moved to public cloud infrastructure entirely. Due to this reason, many organizations adopt a *hybrid cloud* approach, where on-premises servers are used together with cloud-based solutions. In a hybrid model, critical functionality, databases, and security sensitive information stay in the infrastructure that is fully controlled, thus trusted. Only noncritical operations that usually require significant computing power are offloaded to public infrastructure. A hybrid cloud approach is a trade off between a security and performance. From a performance point of view, it may be beneficial to position some services closer to end users to enhance the user experience. When services are located farther away from users, it can result in greater network latency and slower data transmission speeds. By positioning services closer to the end user, these issues can be mitigated, resulting in faster load times and more responsive interfaces, and an overall better user experience. However, preparing the private infrastructure to handle sudden traffic spikes can be more challenging. Such an infrastructure usually has limited resources and may not be able to handle large number of requests. This can lead to downtime or slow performance during peak usage periods, which can negatively impact user experience. As a result, it is important to carefully consider the trade-offs between optimizing performance through infrastructure placement and managing the risks associated with traffic spikes in a private infrastructure. Hybrid cloud models can help protect a company's most valuable assets, which typically include information and data. Data protection is a critical concern for many organizations. It is assured by confidentiality, integrity and availability that are considered the most important information security concepts. They create a "triad" framework (CIA) that sets directions in the development of security policies for organizations. Confidentiality ensures that only authorized individuals can access information. Availability ensures that data is available and accessible when requested. Integrity guarantees that the system is functioning as intended and that its data and operations are accurate, complete, and secure from unauthorized access or modification.

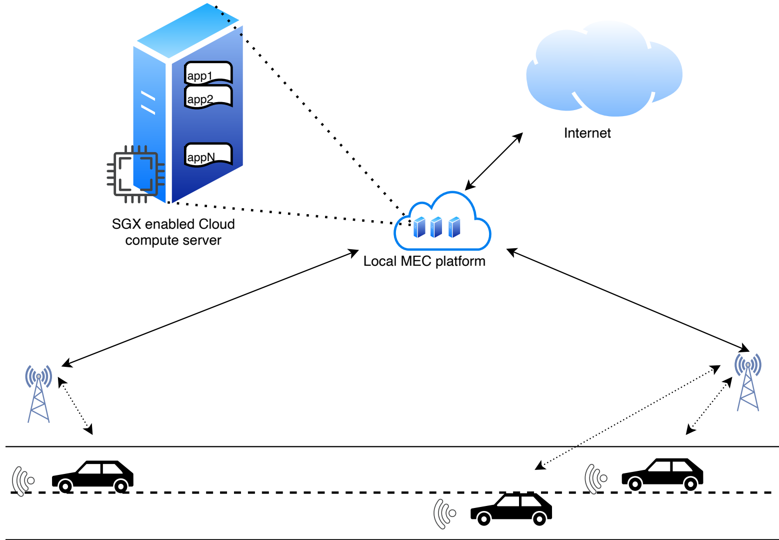


Fig. 2.1: Vehicle-to-Vehicle communication for self-driving cars

Fig. 2.1 presents a use case for which the integrity (and other security pillars) of the entire system are critical. Such vehicle-to-vehicle (V2V) deployment is achievable with 5G edge computing platforms, which opened new space for innovative applications. Service providers can take advantage of that infrastructure and public cloud services to enable V2V applications. Such applications can help to optimize traffic and support self-driving capabilities. A fast-running car needs to know what is the situation on the road ahead in a fraction of seconds. The vehicle can also use real-time generated and realistic maps that reflect the current situation on the road. Such maps can be enhanced by less critical information fetched from the Internet as a vehicle is moving. For such services, a mistake or misfunction is not acceptable. Therefore, the entire solution must be deployed in a reliable and secure manner. To achieve this, the application vendor must introduce strict security measures to ensure that its mission critical application is not compromised.

2.1 Integrity

Integrity plays a critical role in protecting files against unauthorized modification, hence ensuring that applications perform correctly. A mechanism enabling that

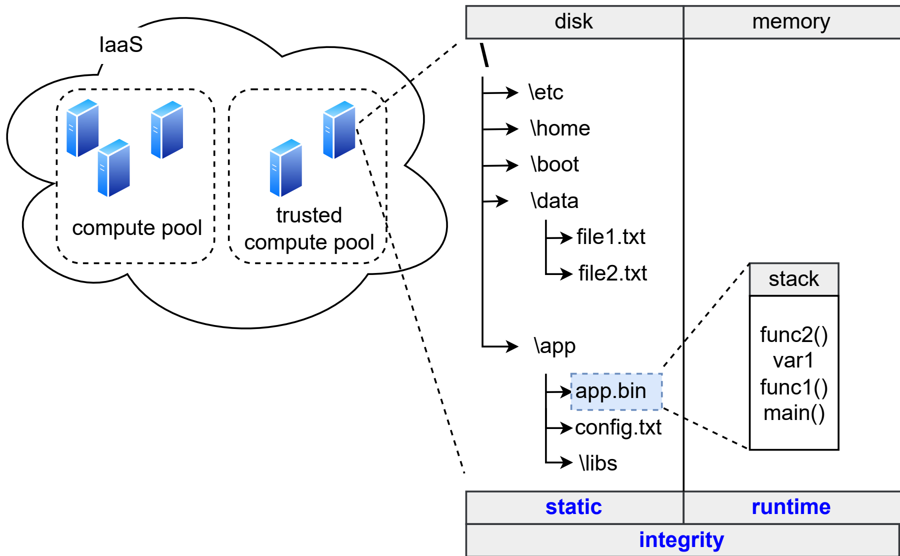


Fig. 2.2: Static and runtime integrity

property should at least detect if the application has been modified. The security policy that prevents modifications and detects changes is a fundamental concept of trusted software. But how do we know that the application is trusted? The answer is to establish a chain-of-trust through all the layers to root-of-trust, i.e. hardware, which is an immutable component critical to ensure security. The chain usually starts from a unique serial number or a key pair burnt into the hardware by a manufacturer. It is assumed that such a design is impossible to compromise. These components are fundamental elements of *Trusted Computing Base (TCB)* that enforce system security. However, a virtualization layer adds complexity to chain-of-trust. On the one hand, the root-of-trust is based on an immutable hardware [64], but on the other hand, it should be virtualized and available for guest virtual machine (VM) running on a physical machine. There are solutions that do not rely on hardware root-of-trust and provide software implementations [10, 12] instead of physical equivalent. Nevertheless, they cannot guarantee a required level of security [16, 140]; thus, we do not consider such models.

Usually, integrity is evaluated only on the basis of static resources, such as files. This type is called *static integrity* as presented in Fig. 2.2. Static integrity should ensure that files are not manipulated by unauthorized entities, but it does

not consider runtime operations. This means that a system should use binaries, libraries, and data files with approved and proper content. There is no single solution that can guarantee static integrity for various platforms [33] (e.g Arduino, embedded platforms or server), architectures (e.g. x86, Sparc) or deployment models (bare metal, cloud). In our proposal we employ Intel SGX that can be used for a bare metal and also for virtualized workloads. We introduce this technology in Section 2.2 explaining key concepts used in our solution.

Modern attacks against a sequence of in-memory application invocations, called a *control-flow*, do not change the application files. Instead, they modify the memory, so that the static integrity remains intact. Therefore, the conventional approach to integrity aspects based on a remote attestation is not sufficient. In addition to that, there is a wide range of computing devices, including desktops, IoT devices, and server solutions. They are based on different architectures and use various software stacks. This diversity makes them difficult candidates for any attestation based on static file measurement. A program execution flow plays a critical role in assuring that developed software behaves as designed and expected, i.e. in assuring *runtime integrity*. The problem arises when a sequence of internal and external calls is not correct and can lead to unexpected behavior. Languages such as C or C++ enable fine-grained control in many aspects of programming, like memory allocation or jumps. They also allow unsafe functions, leaving more space for potential misuse. The structure of popular system libraries, for example `glibc`, is also well known. Therefore, attackers can discover the memory addresses of library subroutines. Then, they can prepare a stack buffer overflow by replacing memory pointers in a way that it will not crash the stack. All this means that a programmer has to manage security aspects to ensure proper resource allocation in order not to allow code abuse by adversaries. Even with all best practices implemented, other components (e.g., an operating system or shared libraries) that also take part in program execution are beyond the control of the programmer. Control flow-related problems have been partially addressed by the software-based mitigation methods. Some of them have been adopted by the industry and are now available in Operating System (OS) distributions.

For the sake of illustration, we refer to *stack canaries* that can detect a stack buffer overflow before code execution happens. With this method, an integer number is placed in a memory directly preceding the stack return pointer. To take control of the process by overwriting the stack return pointer, the integer must also be overwritten. The value of this canary integer is compared against the original value before the return pointer on the stack is used. This method can protect against conventional buffer overflow attacks. To make it harder for the attacker to find key memory addresses that point to program data structures or functions, an Address Space Layout Randomization (ASLR) [118] can also be used. With ASLR, the address space is randomly arranged and the attackers have

to put more effort to discover the memory pointers they are looking for. This is due to the unpredictable address space, making it more difficult for attackers to guess the location of the code that will be executed. Additionally, certain memory areas can be marked as non-executable to prevent the code located in those areas from being executed. This technique, called Data Execution Prevention (DEP), leverages the CPU `NX` bit to mark some of the memory areas as executable or non-executable. When the memory area is not executable, a code cannot be run from that memory page and any attempt will result in memory violation. However, such basic protection methods are not sufficient to mitigate modern exploit techniques that leverage Return-Oriented Programming (ROP) or Jump-Oriented Programming (JOP) and can bypass various defenses [102]. With the ROP attack, the attacker leverages a direct return instruction (RET) to construct a sequence of code instructions that the programmer did not assume. This instruction gets the address of the next instruction from the stack and transfers the execution to a new location. The attackers are aware of the memory addresses of library subroutines. In consequence, they can prepare a stack buffer overflow that contains a chain of subroutines from the system library that can perform the intended malicious action. An ROP attack is difficult to discover because the attackers do not change the application code. They just create a sequence of calls ending with a legitimate return instruction that modifies the existing stack content. On the other hand, a JOP attack is similar to ROP, but affects indirect invocations. Such instructions do not explicitly include a destination address, instead, they have a pointer to a place in the memory where it is stored. Here, the attacker creates a chain of indirect jumps to perform the intended actions that were not implemented by the programmer. This leads to a variety of code-reuse attacks, where the attacker abuses the existing application code to perform operations not designed by the programmer. With this technique, adversaries exploit program vulnerabilities to hijack the control flow. Therefore, we propose possible countermeasures based on Intel CET and a comprehensive approach to the program flow violations, next to static integrity verification.

The lack of control flow information is the main obstacle to a strong defense mechanism against sophisticated runtime attacks [80]. The Control-Flow Integrity (CFI) techniques are responsible for detection and mitigation of various attack on the flow of the program [103]. A Control-Flow Graph (CFG) represents all paths that might be traversed during program execution (as presented with a sample graph in Fig. 2.3). Typically, CFI solutions consist of two phases: the *offline analysis* phase, where such CFG is constructed; and the *runtime* phase, where CFG is enforced. The goal of the CFI-related activities is to restrict application calls not assumed by the programmer and to allow only for valid flow transfers, i.e. the operations intentionally implemented and described by the CFG. A forward control flow, represented by a forward edge in the CFG, occurs when the control

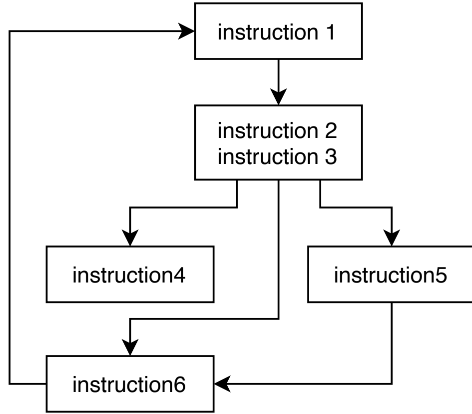


Fig. 2.3: Sample control flow graph

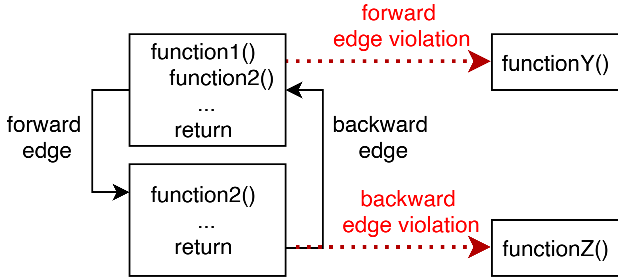


Fig. 2.4: Forward and backward edges.

is moved to a new location in a program, see Fig 2.4. The forward instructions consist of direct or indirect *calls* and *jumps*. They transfer control to another memory location. A direct call or jump occurs when the target address is constant and can be discovered statically. In contrast, the target address of the indirect instruction is dynamic and computed during execution. A backward control flow, represented by a backward edge in the CFG, occurs when the program execution returns to a previous location. Most CFI architectures focus on indirect instruction targets, assuming that direct targets are protected against modifications by DEP. Indirect, i.e. dynamically created, instruction targets provide flexibility, but can be modified by the attacker and they can, in consequence, lead to software abuse.

The CFI mechanism tries to statically determine a list of allowed indirect targets and can detect when a code pointer is outside of that allowed list.

Most of the CFI solutions are software-based. They require code instrumentation to capture sensitive flow information during program execution. They rely on another piece of software to verify flow integration in a runtime. They can also work offline and analyze execution traces produced by the application during execution. In this case, the attack can be discovered after it happened. Some algorithms, called hardware-assisted, can leverage hardware to gather trace information and then use software to process it and discover flow violations. The quality of a CFI algorithm strongly depends on the accuracy of a constructed CFG, therefore, a CFG should accurately represent all possible flow transfers. There are two approaches to CFG reconstruction. Firstly, a program source code can be statically analysed to recreate all flow transfers [17, 85, 126]. Burow et al. [20] analyzed the precision of a CFG reconstruction using various static methods. The authors noticed that increasing the precision of code analysis has a negative impact on the performance of the CFI algorithm. Secondly, a CFG can also be reconstructed during the offline phase from traces generated during program execution. Such traces represent atomic operations [138, 142] within a program and are not comprehensive enough to reconstruct the accurate CFG necessary for a reliable CFI mechanism. Assuring 100% code coverage and enforcing CFI for every possible application flow is still challenging and requires a fresh approach for the CFI-related problems. The fact that a solution can protect some of the functionalities against application flow attacks is not sufficient. It should provide real-time protection for every piece of code that is executed in the environment, including software that is out of our control (like libraries or the operating system). Code instrumentation that introduces a risk of misbehaving should be avoided. Additionally, control flow enforcement has to be easily applied without extra code development and with an acceptable performance overhead. With CET one does not have to worry about CFG inaccuracy. Our solution does not rely on a CFG and does not introduce any extra complexity. Instead, it leverages existing components, i.e. a CPU and a hypervisor supporting a required set of instruction. In the two following sections we introduce hardware technologies that we employ for our solution.

2.2 Intel SGX

Intel SGX is a set of hardware-based security extensions built into Intel CPUs that provide a secure execution environment for software applications. SGX provides a secure *enclave*, which is marked as a green block in Fig. 2.5 that isolates sensitive code and data from the rest of the system. This enclave is a secure container that can store sensitive data and code, and can only be accessed through a set

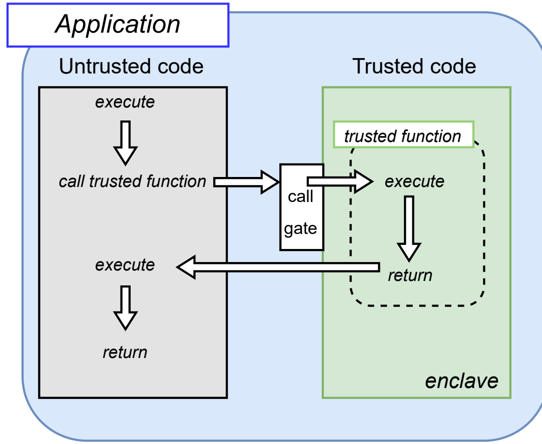


Fig. 2.5: SGX concept

of secure instructions provided by the SGX architecture. The enclave is created by the application developer and is *sealed* with a unique encryption key that is derived from the CPU and specific to the enclave. This key is used to encrypt the contents of the enclave, ensuring that the data and code within it cannot be accessed or modified by any other process, including the operating system. When an application requires access to the enclave, it must first go through the attestation process, see grayed out area in Fig. 1.1. Once the application has been attested, it can send data and code to the enclave for processing as presented in Fig. 2.5. The enclave will then execute the code and perform any necessary computations on the data, ensuring that it is protected from unauthorized access or modification. Afterwards, the execution is transferred back to the unprotected part of the application.

A sealing mechanism built into SGX allows developers to protect sensitive data within the enclave, even if the enclave is moved to a different system or if the system is restarted. This mechanism works by encrypting the data within the enclave with a sealing key, which is derived from a combination of the CPU and the data itself. The sealed data can then be stored on disk, where it can remain protected by the sealing key. To unseal the data and access it again within the enclave, the developer must provide the same sealing key that was used to seal the data originally. This ensures that even if an attacker gains access to the sealed data, they will not be able to decrypt it without the correct sealing key and access to the same CPU. Furthermore, the sealing mechanism includes a set of integrity checks that ensure the sealed data has not been tampered with or modified since

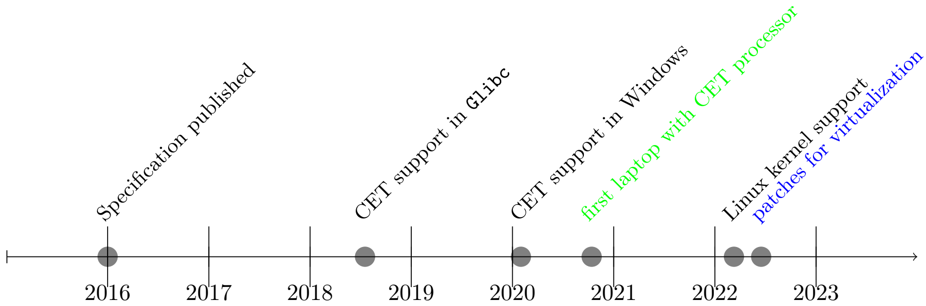


Fig. 2.6: Intel CET availability

it was sealed. If the integrity checks fail, the data will not be unsealed, providing an additional layer of protection against attacks.

The process of developing secure SGX enclaves is simplified by a tool provided by Intel, called *Edger*. To leverage this tool, a developer has to use Enclave Definition Language (EDL) to specify the types of data that can be passed between the trusted and untrusted code, as well as the entry points for the trusted code that can be called by the untrusted code. Enclave Edger uses this definition to generate the interface code (a call gate in Fig. 2.5) that is used to communicate between the trusted and untrusted code. The generated code includes the necessary function prototypes and data structures that enable communication between the untrusted and trusted code. This simplifies the process of developing SGX applications, as the developer can focus on writing the trusted code within the enclave and rely on Enclave Edger to handle the interface code. In Chapter 4 we detail on how we applied described process in our solution.

2.3 Intel CET

Intel announced hardware support for CFI, called Intel CET, in June 2016. While waiting for real hardware support for the technology, the developers had enough time to introduce CET support in mainstream applications and operating systems. However, four years later, at the time when the technology started to be available in the Tiger Lake processor family (marked green in a Fig. 2.6), still most of the software is not ready to take advantage of CET-based protection. The adoption of technology has accelerated with the availability of processors in the market and in 2022 the protection was officially available for Linux operating system. With the huge interest in developing CET related applications in a virtualized infrastructure, necessary support was experimentally added to support virtual

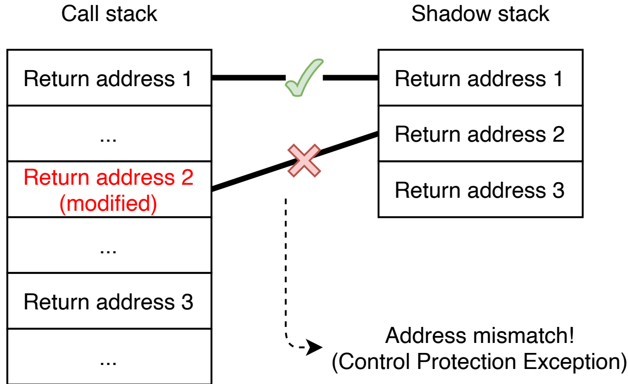


Fig. 2.7: The shadow stack concept

deployments (blue in a in Fig. 2.6). We use this emerging code base to build our solution.

From a technical point of view, Intel CET is a set of CPU instructions that implement hardware CFI that enforces protection against ROP /JOP types of attacks. It extends the processor instruction set with two new features:

- shadow stack,
- indirect branch tracking.

The goal of the *shadow stack* is to prevent attackers from a return address replacement. The mechanism is based on the application of the second stack (as presented in Fig. 2.7), which is hardware-protected against any modifications from the software. During program execution, a call instruction pushes the return address pointer to both: the regular and the shadow stack. When a return instruction is called, it gets two addresses from both stacks. If they do not match, it means that the return address in the legacy stack was modified. In such cases, the Control Protection Exception is raised, and the program execution is stopped.

Another crucial mechanism, *indirect branch tracking*, is based on the new `ENDBRANCH` instruction that is used to mark valid function entries for indirect calls and jumps in the program. It is added by the compiler as the first instruction in every subroutine or valid function. As an example, Fig. 2.8 shows disassembled functions `printf` and `main`. It can be seen that the first instruction in both functions is `endbr64`, which is the `ENDBRANCH` instruction for the 64-bit program. This is the only valid entry point: any attempt to call any other instruction

```
0000000000001080 <printf@plt>:
    1080:       f3 0f 1e fa             endbr64
    1084:       f2 ff 25 3d 2f 00 00    bnd jmp *0x2f3d(%rip)
    108b:       0f 1f 44 00 00          nopl  0x0(%rax,%rax,1)

0000000000001189 <main>:
    1189:       f3 0f 1e fa             endbr64
    118d:       55                      push. %rbp
```

Fig. 2.8: The indirect branch tracking example

will generate an exception. `ENDBRANCH` instructions do not affect the execution of a program and are transparent for the processors not supporting Intel CET technology.

To take advantage of hardware CET support and target validation for the `ENDBRANCH` instruction, the application must be protected. By a 'protected application' we mean the application that was compiled with a necessary compiler flag. As an example, for the `gcc` compiler we used the `-fcf-protection` option with the **CET-enabled** scenario as described in the following section.

2.4 Remote attestation

The most popular approach of assuring integrity for systems that offer compute capabilities is *remote attestation* [33]. *Remote attestation* is a building block for trusted computing [19]. It allows one to determine the level of trust and integrity of the platform [7, 10, 41, 89]. Remote attestation is enabled by the *Trusted Execution Environment (TEE)*, which provides the trusted system components that form the basis of security [83, 84]. In this process, the TEE measures the software and hardware components of the system and generates an attestation report. Based on that report, a remote attestation qualifies hardware and software security protection mechanisms that a system includes to protect the data confidentiality, availability, and integrity. Remote attestation becomes critical when processing is moved to the cloud, and the application vendor uses a leased physical infrastructure. A security risk that is a consequence of such shift should be mitigated by the application vendor by using software verification before any sensitive information is processed.

Different methods have been proposed for remote attestation of a bare metal [5]. Based on the verified objects, they are divided into the following two categories: (a) static integrity verification, (b) runtime integrity verification.

A static integrity verification process (for static resources) usually boils down to a simple checksum comparison. A new checksum of a static file, called *measurement*, is compared with its original value. If they do not match, it means that a file was modified. A typical remote attestation process comprises the following steps:

step 1 the attestation software measures static files,

step 2 an integrity report is sent to a remote verifier,

step 3 a verifier evaluates integrity of the remote system.

One of the challenges for the IaaS model, where the infrastructure is owned by another entity and host resources are shared, is to ensure that the attestation process that generates an integrity report (step 1) is not affected by intruders. Theoretically, an attacker can manipulate the measurements or the local attestation software, so that a remote *verifier* cannot discover any abnormalities. To avoid that, the attestation software should also be attested before it is used. Moreover, following this way of thinking, hypervisor and host operating system should be also verified. This boils down to the fundamentals of software security, where chain-of-trust starts with an immutable hardware component that is impossible to compromise.

Another challenge in a remote attestation process is secure communication with verifier (step 2). In order to establish an SSL/TLS channel between a client and a verifier and mutually authenticate both parties, the endpoints must possess encryption keys. Providing secure storage that can be used to store a private key inside a virtual machine may require the use of hardware security modules. This entails a variety of problems, as those modules cannot be easily virtualized.

The majority of solutions belong to the first category, i.e. static integrity verification, and detect the presence of unwanted software in a file system by checking the integrity of static files. A solution for attestation based on file integrity verification was originally proposed by TCG. They also formalized an architecture for a TPM module that is widely used in such solutions. Static integrity verification was sufficient measure for most of the threats. Nowadays, with the growing number of more sophisticated attacks, it is required to complement this with runtime integrity (also known as 'dynamic integrity'). Only both, i.e. static and runtime integrity, can ensure overall application integrity and enforce proper behavior.

Runtime integrity should ensure that the application executes only the actions designed and predicted by the programmer [37, 98]. From the security perspective, this implies that a runtime integrity system should enforce only intended behavior, or — in other words — enforce its runtime integrity. For clarity, integrity enforcement should block any unwanted action, while integrity verification should evaluate integrity after the event occurred. A mechanism called a *runtime*

attestation was proposed by researchers [39, 46, 106] to assess runtime application integrity. The nature of the object being verified implies new challenges. As an example, we consider a memory stack that keeps the order of method execution (control-flow) and local variables. First, the content of a stack changes after every execution of a program instruction. This change should be recognized and its integrity should be measured immediately. In this case, the runtime attestation software should be aware of the proper stack content, which is challenging for highly dynamic and volatile objects. Second, the large number of values that can be measured, requires an efficient real-time attestation solution.

Researchers and security engineers tend to focus on static or runtime integrity [61], making the assumption that the rest is provided by another solution. However, in practice, such an approach often involves major challenges in the complexity of the architecture, which makes it infeasible for virtual deployments. Those challenges are described in the following section.

2.5 Challenges for remote attestation in a virtualized infrastructure.

The most widespread solution to ensure the integrity of the virtual machine is based on Virtual Trusted Platform Module (vTPM). In this section, we describe this approach and show that it does not address some critical aspects recognized by the industry.

A vTPM [99] architecture describes the building blocks of the infrastructure, which expose the TPM capabilities to virtual machines. The module creates a private dedicated TPM instance for each virtual machine running on hardware that enables trusted computing capabilities, such as secure storage, sealing, or attestation. A chain-of-trust is used to validate each component from the hardware root entity up to the application level. Virtual TPM Manager associates a single hardware TPM module with the individual vTPM instances as shown in Fig. 2.9. Virtual TPM Manager uses certificates and asymmetric cryptography to secure the communication and authorize entities in a created chain-of-trust. In the proposed model, *Platform Configuration Registers (PCR)* are divided into two parts. The lower registers keep hardware measurements for the host and are read-only for the guest VMs. The upper PCR are responsible for filesystem measurements and are maintained individually for each guest virtual machine. Such an approach enables attestation capabilities for the VM and lets the verifier acquire knowledge about measurements not only for the virtual machine but also for a host environment.

One of the major challenges is to meet the requirements related to complex VM operations. A virtual machine can be relocated between clouds X and Y (ref. Fig. 2.9), scaled horizontally or vertically, or restored based on a snapshot from the

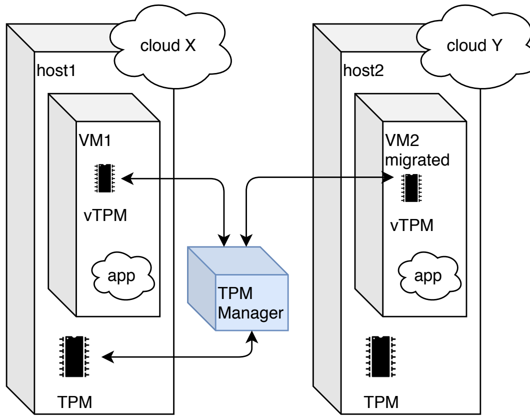


Fig. 2.9: vTPM architecture in a multi-cloud environment

past. This flexibility also means that a virtual machine can be restarted on another piece of hardware and even in another physical location owned and maintained by another entity. Deployment of an application in a trusted environment should be insensible to the mentioned operations. Thus, it is necessary to track TCB, because the trust that was established in the initial environment has to be transparently reestablished again. Moreover, since vTPM may store data, such as encryption keys or certificates, externally to the VM's memory, these data have to be migrated together with a VM. The migration process should guarantee confidentiality, integrity and availability for vTPM and the migrated VM when deployed to a target destination.

The aforementioned requirements are difficult to meet in multi-vendor cloud environments in a way that the application vendor can control the VM security aspects across all cloud environments in a similar, uniform fashion transparently to the VM. For the application vendor, it means that the attestation capabilities are very limited and service provider lock-in cannot be avoided. This also means that the application cannot be deployed in desirable infrastructure without a security trade-off. As a consequence, performance and overall end-user experience are likely to be suboptimal, because the user is redirected to a datacenter meeting application requirements, not necessarily the closest one.

Considering described obstacles, we propose an architecture for static and runtime integrity verification, as shown in Fig. 2.10. Our model is based on hardware modules that can be exposed to a virtual machine. Those modules

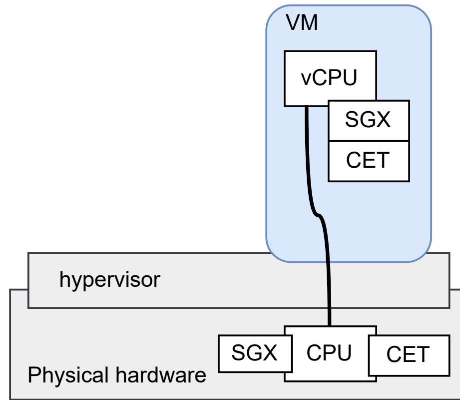


Fig. 2.10: Proposed, simplified architecture for VM static and runtime integrity verification.

(SGX and CET) are built-into a processor, therefore, do not require additional software components other than a hypervisor to run. At this stage, we introduce a concept that is described in details in the following chapters.

3 Related Work

The spread of the Internet has significantly increased the number of services provided remotely. For such services to be reliable, additional security measures had to be implemented. One of them is remote attestation, which has become a widely used method for verifying the integrity of a remote system. Remote attestation based on TPM for many years has been a sufficient method to protect against security threats to integrity. Since then, systems have become much more complex, and virtualization has introduced new ways to run applications. To follow this development, an approach to a remote attestation process also has to be re-evaluated. In the sections below, we discuss solutions for static and runtime integrity separately, and afterwards we analyze a security of our proposal. To the best knowledge of the authors, there is no solution that can address both aspects comprehensively.

3.1 Static attestation

For conventional bare metal deployments, a creation of trust for running applications can be achieved by leveraging built-in hardware modules. Here, we can enumerate, for instance, TPM [129], TrustZone [137] isolation or processor capabilities such as SGX [34] instructions. More details about each approach are provided below.

With TPM, proposed by TCG [66], secret keys are stored on a dedicated chip and separated from the filesystem. The built-in PCR can be used to save measurement values, i.e., digitally signed cryptographic hashes of files. They serve to prove integrity. With this approach, Trusted Computing System (TCS) [62] based on a TPM creates a root-of-trust for applications based on hardware modules [104], where each executable starting from BIOS is verified

before execution. In this case, it is necessary to use a TPM-aware boot loader that can measure the executable and configuration of the kernel before it is executed. The captured PCR measures can then be validated by a remote verifier to prove the integrity of the application. One of the most widespread solutions based on a TPM system is the Integrity Measurement Architecture (IMA) subsystem [107]. Using the TPM module, it stores SHA1 values of measured static files in the PCR registers, preserving a chain-of-trust from the hardware module to single files. IMA formed a foundation for other techniques that attempt to address many limitations. For instance, TPM has a limited number of PCR registers, measurement is dedicated to static files and not dynamic software executions, it cannot deal with file upgrades, etc. Other, property-based attestation models [26, 70, 71, 105] assume that the attestation of the entire platform should not depend on the specific versions of the software or hardware. They introduce a mapping between software components and properties. It means, for example, that a software update will not fail the attestation as long as the property is trusted. Although more flexible than IMA, this model is not widespread. The major concern is that a proper mapping mechanism between static system objects and the properties leaves space for abuse.

Another method leverages a processor TrustZone [11] to provide virtual isolation between secure and insecure spaces [100]. System measurements stored in the secure space are protected against software running in the unprotected area. However, TrustZone itself does not provide protected storage or built-in authentication when the application communicates between these spaces. Thus, the solution is vulnerable to attacks when transferring data between them. Processes running in a secure area share the same memory, so attestation software is threatened when other software has vulnerabilities. For these reasons, We do not focus on TrustZone. The CPU isolation feature that TrustZone provides is not sufficient itself [76]. Virtualization support is in research [58, 101]; therefore, making it available to virtual machines is challenging [58].

The SGX-based attestation allows a remote party to gain confidence that the intended software is running within an enclave on an Intel SGX-enabled platform. Intel SGX supports the two types of attestation as an integral part of the architecture: local and remote attestation [34]. The local attestation allows two enclaves on the same platform to attest to each other. Enclaves assess their identities and verify hardware reports to confirm that they share the same platform. This is not sufficient when the enclaves are distributed and the verifier has to assess them remotely. With the remote attestation, one can remotely verify that the enclave runs on a genuine SGX and assess a firmware security level. When the remote attestation is successful, the enclave can act as a hardware root-of-trust and can be used as an execution environment for sensitive operations. As an example, Knauth et al. [67] leverage SGX remote attestation to enhance

the TLS protocol with platform-specific information. They use SGX enclave as a hardware root-of-trust, opposite to a list of root certificates built-in the browser. With this approach, the client and server are mutually authenticated and can start encrypted communication.

The aforementioned attestation methods rely on hardware security modules or specific software to run. Together with new technologies (such as IoT, embedded devices, or virtualization), the approach to attestation must be redefined. For small IoT devices, that are usually cheap and limited in terms of hardware, one may choose collective attestation [7] or software-based remote attestation [23, 116, 141] that does not guarantee the same level of security. Additionally, the network connectivity characteristics [56, 123] can be used for fingerprinting. This fingerprint should not change if the device is not relocated and serves the expected load patterns. A fingerprint can then be included in the attestation report for integrity evaluation. Small and resource-limited devices, e.g. IoT, embedded devices or Arduino platforms, are often deployed in swarms. They cannot handle expensive computations for the attestation individually; instead, they require a different approach. Swarm attestation models [6, 8, 14, 69] minimize communication overhead, computations, and power consumption. Such models can be considered for swarms with heterogeneous devices, i.e. with different architectures and access methods, and also for devices deployed in dynamic networks. A minimalistic approach to remote attestation combines both hardware and software components, as defined by Francillon et al. [48]. They provide exact properties needed to ensure remote attestation, i.e. exclusive access to a key, no leaks, immutability, uninterruptibility, and controlled invocation. These properties were assigned to a set of system features that together achieve the required attestation. Those system features are based on specific platform independent hardware and software and can provide the attestation for any underlying architecture.

Collective attestation models are designed to serve a large number of nodes and are not affected by potential bottlenecks caused by a single verifier. Song at el. [121] proposed a scalable solution for cloud based deployments that can handle any practical number of nodes. Instead of verifying each node individually, they constructed an attestation relationship tree. Each parent in a tree structure is responsible for attestation of its child nodes. The aggregated attestation report is then passed to the verifier for evaluation. With such a schema, the verifier is never overloaded. On the other hand, this method increases the complexity of the solution. The authors of [121] assumed that during attestation the topology cannot change and the nodes must remain reachable.

The described solutions are not good candidates for virtual deployments. Such solutions often need to access the hardware root-of-trust, but in the cloud infrastructure, such hardware root-of-trust is not available. Because of that, virtual deployments, which can run similarly to bare metal software, offer much

less security. It is expected by the industry that security for sensitive applications should not suffer when migrating to cloud infrastructure. The most popular model that leverages hardware root-of-trust and can be applied for cloud infrastructure is based on vTPM that enables TPM functionality inside a VM. It is described in details in Section 4.1. From a security perspective, vTPM design is vulnerable to powerful adversaries. Ozga et.al. proposed *TRIGLAV* [93] that employs IMA and SGX to mitigate problems with vTPM design recognized for virtual infrastructure. Their solution is claimed to provide VM integrity for public clouds with acceptable overhead (<6%). Although vTPM based solutions may provide the required functionality, they suffer from another problems that impacts the adoption in the industry. They are related to the complexity and the ownership of the solution, often ignored by researchers. A cloud infrastructure that is already complex is maintained by the cloud provider. An application vendor in the IaaS model is responsible only for their VMs and should have limited trust in any other components of the infrastructure. This includes any attestation solution that is based on a hardware owned by the cloud provider. Then a question arise: can one trust in the third-party attestation solution based on a component he cannot control? From this reason, it is visible in the industry that they decide to build their own remote attestation process for fully controlled on-premise private clouds. Our model is based on a processor directly exposed to a virtual machine as opposed to a hardware module on a host motherboard. It completely eliminates the problem of shared responsibility and requires small changes in the cloud software stack. The proposed solution is described in Chapter 4.

3.2 Runtime attestation

Advanced CFI techniques can ensure the sequence of executions in a program flow protecting against more primitive attack patterns, but the CFI implementations are not mature enough to rely solely on them as highlighted in the following section.

There are several proposals for runtime protection in the literature. In their example, we discuss the most important limitations. One of such solutions, called the Scalable Runtime Remote attestation (ScaRR) schema [128] utilizes the control-flow attestation. The authors claim that it is suitable for virtual machines running in a cloud, including the most popular public clouds. ScaRR requires code instrumentation, i.e. adding extra instructions to the application code to monitor program behavior. It generates a CFG for the monitored application in the offline phase. Since analysing complete program execution paths in a CFG can be expensive, the authors proposed to select only meaningful sub-paths for faster processing. The authors claim that ScaRR has a runtime attestation speed 10 times faster than other existing solutions, but it requires high network throughput

that may significantly slow down the verification speed and impact real-time verification. The solution assumes static remote attestation based on hardware modules like TPM for prover and verifier, but the document does not mention how a chain-of-trust can be established in a cloud environment. TPM-based remote attestation for cloud deployments is challenging due to its complexity and limitations.

A Control-FLow ATtestation (C-FLAT) [3], dedicated to embedded devices, is also based on a control-flow within the application. This solution also requires code instrumentation to extend a binary with C-FLAT instructions. Such instructions allow for capturing operations like branches or function returns, and create control flow paths during execution. Here, prover presents a report of the application flow to verfier that takes advantage of CFG to assess the runtime behavior. The authors also notice that accurate CFG generation is an open problem, which can be complex for general-purpose computers. An idea that does not require code instrumentation is presented by Dessouky et al. [43] with their Low-Overhead Control Flow ATtestation (LO-FAT). It is faster because it leverages processor capabilities. It captures the source and destination address of each branch from the processor and computes a cumulative cryptographic hash of the executed path. This is done to avoid sending every instruction to a verifier, thus saving compute and network resources. Inspired by the described C-FLAT and LO-FAT, Zeitouni et. al presented runtime ATtestation ResilIent Under Memory attacks (ATRIUM) [143]. In addition to control-flow measurements, the authors also evaluate instructions being executed by the attested program. By coupling them together with the control-flow measurements, their solution can prevent from attack called Time of Check Time of Use (TOCTOU) [42]. Virtualized deployments are not susceptible to such threats, because an adversary does not have physical access to the prover device. This means virtualized deployments do not benefit from ATRIUM protections, which makes the solution better suited for physical devices.

Another approach based on control flow information is presented by Koutroumpouchos et al. [71]. The authors noticed that analyzing every CFG path is practically impossible. They focused only on the paths that are sensitive or critical to the application that serves a particular function. With their Lightweight Control-Flow Property-based Attestation (CFPA), a critical subset of a CFG must be identified in advance and can be updated later. This gives a necessary flexibility to adapt to changing conditions on the application side. The authors assumed that a static attestation is applied to their solution and identified a number of security-related challenges that need to be addressed. One of them comes from the fact that prover and verifier should establish a TLS-based communication to exchange information. It is unclear how to guarantee secure key management, since the prover runs in the untrusted environment. Another

open issue is the fact, that a verifier should also be protected against control-flow related attacks, unless it runs in a fully controlled bare metal environment. Otherwise, a trusted component to attest a verifier is yet to be identified.

A method known as Remote Dynamic Attestation System (ReDAS) is presented by Kil et al. [65]. It proves the integrity of certain properties of the application runtime. Those properties uniquely reflect the runtime, and can reveal an integrity violation when modified. As an example, the address returned by a function is verified to point to the expected location. The evidence is stored in a PCR built in TPM to protect measurement values from tampering. This design reveals the limitations for IaaS deployments, related to the availability of hardware security modules. Moreover, ReDAS uses system calls to verify system integrity and may not detect runtime misbehavior between them. With such limitations, the average performance overhead was 8% with promising effectiveness.

Majority of traditional attestation models, as noticed by Haldar et al. [55], is based on a fact that a particular program binary was run. In that case, trust is confirmed by a verifier who knows what code a remote system should run. Then the strong assumption is made that a binary should behave as expected. This is not a verified assumption and it is based purely on a trust. Haldar et al. highlight that the real intention of an attestation is to verify that a remote software behaves as expected. The authors propose a semantic remote attestation. As an example, a Java Virtual Machine (JVM) is a type of language-specific virtual machine that can execute platform-independent application code. It can then monitor the execution of the application running within JVM. Additionally, a trusted JVM can attest high-level program properties, such as method signatures or class hierarchy, without knowing what the program is actually doing. With a semantic attestation, the authors assume that the operating system and the trusted virtual machine have to be statically attested using one of the existing methods. This solution is applicable only for applications that require JVM to run and hence is limited in scope.

Similar assumptions to attestation are presented by Alam et al. [4]. They noticed that a proper application behavior, called a trusted behavior is desired, and not a fact that a binary was run. However, a trusted behavior is not defined. Instead, application vendors trust that their applications will execute their code properly. In the proposed Model-based Behavioral Attestation (MBA) framework, the attestation is not tied to any hardware or software platform; instead, the authors evaluate the behavior of an application described by a policy model. A policy model is based on Usage Control (UCON) [94], a complex usage control model that describes all aspects of possible use cases in the application. In the initial phase, a set of behaviors for the components of a policy model must be specified and formalized, for example, the object A can access the object B under condition C. With the MBA solution, it is expected that a platform being attested

sends a behavior of a UCON policy for assessment. Such captured behaviors are verified with the expected behaviors, and on the basis of the result, the trustworthiness of the remote platform is evaluated.

Many runtime attestation models leverage CFG that describes legitimate flow directions inside the program. Such CFGs can be obtained statically from source code analysis [17, 85, 126], or dynamically disassembling a program binary and using processor traces generated during program execution [138, 142]. Static methods are not satisfactory as they always lead to some over-approximation of possible control flow transfers. This stems mainly from the assumption that all possible conditions in a source code can happen. However, some of them are exclusive, duplicated, or overlapped. This means that the CFI mechanism can be suboptimal due to the processing of not existing flow targets. Burow et al. [20] classified and categorized numerous CFI implementations published in years 2005–2016. They compared the precision of CFG reconstruction using different static methods. It had been noticed that when the precision of the static analysis is increased, the performance of the CFI mechanism deteriorates. On the contrary, the dynamic approach gives different results and usually underestimates possible program flows. The discovery of the flow relies on analyzing the traces of program execution. In this case, some paths might simply be omitted when the program does not execute them. The results of a dynamic reconstruction allow identifying frequently visited program flows because running the same code multiple times will generate similar traces. This knowledge can be used for optimization and profiling of an application in opposition to the static approach which is used mostly for security analysis.

The study of the effectiveness of CFI methods against various attack types has been a subject of research for a long time. Sayeed et al. [112] analyzed 14 recent major software-based and hardware-assisted techniques and evaluated them against popular attack vectors. They show a selection of state-of-the-art attacks that can bypass the CFI protection methods considered. For each of the CFI techniques under study, they identified the number of possible bypasses. On this basis, they classify and assess the security of various CFI implementations. None of the examined techniques protected against all types of control-flow related attacks. Only three implementations [53, 111, 127] proved to resist code reuse attacks such as ROP or JOP. Software-based algorithms proved to be more secure and easier to implement in comparison to hardware-assisted ones [15, 81, 87]. None of the researched protection algorithms provided a full defense, and some of them presented only weak protection and should not be considered by the industry. It was also noticed that the execution time increased in the three cases [29, 47, 144] by 50%, which is a very poor level and can disqualify the algorithm.

To enforce CFI, some code transformation or instrumentation is required. This implies a risk of corrupting program functionalities for software that reads its

code at runtime. In such cases, CFI should be avoided, leaving the program parts unprotected. Most studies cover performance and security aspects of the existing CFI mechanisms, but they do not address such software compatibility problems. Xu et al. [139] proposed a set of test metrics called Control-Flow Integrity Relevance Metrics (CONFIRM) designed to reveal code features that are sensitive to CFI protection. With this tool, one can discover code features that can be impacted by necessary code changes required to enforce CFI. The authors used CONFIRM to evaluate 12 publicly available CFI implementations. The results revealed that only 53% of the code of large production systems is covered by CFI solutions, leaving the remaining significant part of a critical component with unaddressed threats.

Described techniques have benefits, but are not widely adopted in the industry. Here are a few reasons for this:

- Performance overhead: Existing CFI methods are mostly software-based. This implies a runtime overhead, which impacts the performance of the application. In some cases, the overhead can be as high as 50%, which is not acceptable for many applications.
- Compatibility and complexity problems: Most of the CFI techniques require code instrumentation and another system to enforce control flow. This leads to incompatibility problems and limited adoption.
- False positives: CFI techniques that are based on CFG are not accurate and can generate false positives. This impacts the usability of the system.
- Limited coverage: CFI techniques only protect against control flow attacks, which are only one of many attack vectors and not necessarily the top priority for the company.

Most solutions enforce runtime integrity by proper control-flow, usually defined by imperfect CFG. Moreover, they assume that a static attestation is in place, and this oversimplification may significantly impact the solution. In the following we discuss a security for our solution. Afterwards we summarize on our discussion highlighting the most important aspects that should be addressed for virtualized deployments.

3.3 Security of technologies used in our proposal

In this chapter, we show different types of attack and defense mechanisms from the literature. This is to gain a better understanding of the existing vulnerabilities and analyze a security of our proposal.

3.3.1 SGX security

In this section, we discuss various hardware vulnerabilities existing on a host or within SGX runtime libraries that can affect usage of SGX. Famous Meltdown [77] and Spectre [68] vulnerabilities that were addressed by firmware updates proved that hardware is not always secure and one vulnerability can affect a huge number of systems all over the world. A recent SGXPectre [24] attack based on vulnerable code patterns inside the SDK runtime libraries is a Spectre attack against the SGX enclave. It was shown that it can compromise the confidentiality of SGX enclaves, as the attacker can steal a seal key and decrypt storage from outside of the enclave [24]. A recent Load Value Injection (LVI) attack (difficult to perform) is dedicated for the Intel processor. The attacker can intercept data from a microarchitectural buffer. So far, this vulnerability has also been patched [130]. All of those severe vulnerabilities are quickly addressed and fixed by applying CPU microcode updates for affected processors. Intel Attestation Service (IAS) that plays a critical role in the enclave attestation process is checking the attestation signatures generated by the CPUs. It can instantly discover outdated, vulnerable processor firmware. Thus, the attestation service is able to decide whether the enclave is trusted and the attestation process should be continued. This mechanism allows controlling and monitoring affected firmware versions as soon as it is reported.

SGX that is a still relatively new technology, is actively used in various industrial and research projects. Several vulnerabilities have been revealed, but fortunately, solutions to mitigate those issues are also proposed. Memory corruption-based attacks against SGX are still not well explored. As an example, code-reuse attacks [18] prove that root privileges are not required to abuse the SGX SDK functionality. A vulnerability described by Arnautov et al. is related to the SGX exception handling as well as the interaction between the enclave and the untrusted application. The suggested mitigation methods are based on hardening for the SDK. This type of attack shows that the application vendors should consider the implications of using the SDK. They should introduce necessary countermeasures to protect their code and make sure to use the release version recommended by Intel.

A side-channel cache-based attacks focused on L1 caches are not easily executable, and a number of defense methods are known [45, 135, 136]. More sophisticated attacks target Last-Level Caches (LLC) which in contrast to L1 cache are shared by all CPU cores. The attacker can co-locate their malicious application using SGX on the same physical host with a victim's enclave. In such a case, both applications share the same physical Enclave Page Cache (EPC) memory. The attack is based on a *prime and probe* mechanism as described by [63, 115] and starts from cache sets discovery. A high resolution timer is used to measure cache access time. Longer response time implies cache miss, and

analogously, shorter time suggests cache hit. In a *prime* part, all cache sets are filled with attacker data and execution time is measured. A victim, executing operations inside the enclave, replaces some cache sets with its data. In a *probe* part, the attacker again hits a cache that had been filled and measures the access time. A longer access time means that the victim overwrites a particular cache space. With this algorithm, the attacker can discover cache sets that the victim's application uses. The memory allocation function is deterministic; therefore, subsequent application executions result in the same cache hits or cache misses for the attacker. This behavior allows the attacker to create a heat map for cache sets that are used by the victim's application. Knowledge about the details of the encryption algorithm and a cache usage pattern allows the attacker to reconstruct a key, as demonstrated in [45, 115]. The authors provide countermeasures that can be applied in the application, operating system or a hardware. For the application vendor, they recommend *bit slicing*. This technique utilizes bit operations instead of lookup tables. Such operations are not vulnerable to cache attacks and should be used in the entire encryption algorithm. Independently of software protections, hardware countermeasures can be introduced by the platform owner. One of them is introduced by Intel, Cache Allocation Technology (CAT). It can effectively isolate data in the Last-Level Cache. Liu et al. [79] proposed a solution based on a CAT that can be applied to prevent malicious software in a co-hosted enclave. Other countermeasures were discussed by Ge at el. [49], who discuss attacks against shared resources also in virtualized environments and specifically focus on shared cache vulnerabilities.

3.3.2 CET security

Control flow related attacks are a class of runtime attacks that aim to hijack the normal program execution. They allow attackers to execute malicious code or perform unauthorized actions even without changing program files, thus difficult to discover. Here we discuss mitigations to the most common types of control flow attacks existing in the literature. These are the ROP attack, JOP attack and data-oriented attack, where data structures are manipulated in memory to corrupt control flow pointers and redirect program execution.

A couple of methods were proposed to protect the in-memory address manipulation; some of them were adopted and are available in the operating system. As an example, stack canaries [36, 131], described in Chapter 2, that are an effective defense against simple buffer overflow attacks. Sophisticated attackers, however, may be able to bypass this protection by using ROP. A shadow stack [31, 38], that is a separate stack maintained alongside the regular program stack can increase the security. It helps to prevent attacks that attempt to overwrite the return address and can effectively defend against ROP. However,

existing shadow stack implementations are software based; therefore they suffer from various problems, i.e. software vulnerabilities and significant performance overhead. The most popular and complex mechanism to mitigate ROP is to enforce CFI by verifying that the target address of a call instruction is valid and matches the expected control flow. Abadi et al. [2] provide a comprehensive overview of CFI techniques and their effectiveness in mitigating control flow attacks. Techniques described by authors leverage imperfect CFG and code instrumentation to prevent code hijacking. Some of them show significant performance overhead to the processing time that can prevent the solution before adoption. An important contribution in this area was done by Carlini et. al [21]. The authors show that the existing CFI techniques may not be sufficient against memory corruption vulnerabilities. They evaluated shadow stacks in combination with CFI techniques and came to the important conclusion that a shadow stack is necessary for security. A presence of a shadow stack prevented before malicious code execution in 50% of tested programs. We leverage a shadow stack built into CET in our proposal.

Although existing CFI techniques can be effective in mitigating certain types of attack, there are still some vulnerabilities that are not fully addressed. Here are some examples:

- Blind spots: The accuracy of CFI strongly depends on the quality of CFG. A CFG reconstruction is challenging; therefore, it may not contain all program paths. In this case an attacker can bypass the CFI mechanism by exploiting a blind spot in the control flow graph and redirect program execution to unwanted code [40]. Our proposal is based on automated code instrumentation and does not require a CFG or any other input information.

- Human error: The quality of implementation of CFI impacts effectiveness. If the implementation contains errors or does not cover all possible corner cases, it may be possible for an attacker to bypass the protections. We rely on existing hardware implementation of the algorithm on a massive scale in a processor. For this reason, human error is negligible.

- Data-oriented attacks: Such attacks modify program data, e.g. a counter variable used in a loop. They cannot be mitigated with existing CFI techniques because they do not modify control flow pointers [30, 57, 86, 90]. Memory safety is the first line of defense. Memory-safe programming languages have mechanisms to prevent memory errors, however widely used C and C++ languages today do not offer such protections. Memory-safety problems exist due to a compromise between security and efficiency. This is due to a performance overhead when memory safety is enforced. There are several mechanisms that focus on data-oriented attacks, but they suffer from high performance overhead [22, 28, 114] or require hardware extensions [120].

- Side-channel attacks: In this type of attack, an adversary can use physical information, such as timing or power consumption, to infer the control flow of a program and extract sensitive information. CFI techniques do not address such attacks, nor do we.

Our proposal for CFI enforcement is based on Intel CET and to our knowledge there have been no known cases of CET being hacked.

3.4 Summary

We examine current methods for runtime attestation and identify crucial aspects for deployments in a virtualized environment. We use them to compare existing solutions in the literature with our proposal. Those criteria are listed in Table 3.1. It should be noted that we analyze only solutions for runtime attestation as an extension to static runtime integrity, which itself is not sufficient. The problem of runtime integrity was addressed considerably later in history. One of the earliest and most famous examples of system integrity attacks is the Morris Worm created in 1988, whereas the first functional ROP technique [117] was presented almost two decades later. We use the following comparison criteria:

- **Static and runtime attestation**. As highlighted in Fig. 2.2, a static attestation is a fundamental integrity check for any system. It should be accompanied with a runtime integrity check for full protection. Some runtime attestations solutions do not mention static integrity, and the others assume only that it is implemented using existing attestation methods and hardware modules. A tick in Table 3.1 means that the solution provides both static and runtime aspects of the integrity.

- **Hardware root-of-trust for a prover**. Any security solution needs an immutable hardware anchor as a base for trusted computing. A tick value in Table 3.1 means that prover can leverage a hardware module to establish a chain-of-trust.

- **Protected verifier**. Most often, a verifier is assumed to run in a fully controlled and trusted environment; therefore, it can be considered trustworthy. Selected solutions describe a protection for a verifer, which makes them better suited to deployments in an uncontrolled cloud environment.

- **Runtime integrity enforced**. All the solutions claim to offer a real-time protection against a runtime abuse. In practice, prover and verifier must exchange messages to verify if the action is legitimate. Furthermore, there is a delay between the occurrence of the malicious action and the time when

Table 3.1: Comparison of solutions providing system attestation

Property	ScaRR [128]	CFPA [71]	ReDAS [65]	Semantic RA [55]	MBA [4]	C-FLAT [3]	LO-FLAT [43]	ATRIUM [143]	SGX+CET
Static and runtime attestation	✗	✗	✗	✗	✗	✗	✗	✗	✓
Hardware *root-of-trust for a prover*	✗	✓	✓	✗	✗	✓	✗	✓	✓
Protected *verifier*	✓	✗	✗	✓	✗	✓	✓	✓	✓
Runtime integrity enforced	✗	✗	✗	✓	✓	✓	✓	✓	✓
Ready for cloud deployments	✗	✗	✗	✓	✗	✗	✗	✗	✓

it was discovered. A tick value in Table 3.1 means that a solution is not affected by this artifact and can enforce real-time protection.

- **Ready for cloud deployments**. It is noticed that the authors used to claim that their solution is suitable for cloud deployments. On the other hand, the solution the authors proposed uses a hardware module, usually a TPM, for security. Typically they do not describe how it is available for a virtual machine. This oversimplification in practice may introduce huge complexity to the proposed solution and in fact can disqualify the solution for cloud deployments. Therefore, we do not classify such cases as ready for cloud deployments.

Apart from the discussed aspects, our solution also meets the requirements recently defined for RA systems in the literature. The significant contribution in this area is presented by Steiner et al. [122] and later by Johnson et al. [61] who classified recent remote attestation systems and structured their problem space. The authors identified five problems that are crucial to address for any RA system. Firstly, a hardware *root-of-trust*. Our **SGX+CET** based attestation assumes hardware root-of-trust that is close to the processor, therefore, it should also be available to any hosted virtual machine. Secondly, the *evidence type*, i.e. static or dynamic data collected for verification. In this thesis, we propose a comprehensive solution for static and runtime (dynamic) integrity enforcement dedicated to virtual machines running in a cloud with CET and SGX [32, 60, 82] enabled hypervisors. Third, discrete at a given point of time *evidence gathering*, or continuous evidence gathering. Our solution is based on a discrete evidence gathering for static integrity verification and allows for real-time runtime control-flow enforcement for the entire software stack. Fourth, a *packaging and verification*, defined as a process of encoding or encrypting evidence to share in a secure way

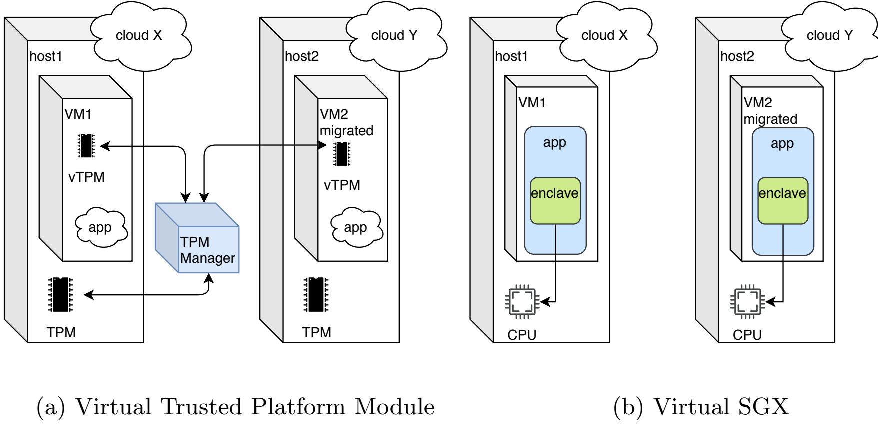
with a verifier. In our solution an encrypted measurement file is received by the Attestation Server (AS) over a secure TLS channel. Finally, *scalability* of the solution. This aspect must be carefully checked for every production solution, therefore we show scaling possibilities of the AS.

4 Static attestation

In this chapter, based on [72], we provide details about our solution for static attestation for virtual machines. We start with an overview of the technology we employ for this purpose.

4.1 Remote attestation with virtual SGX

Intel's SGX is a building block for a trusted computing development. It assumes that everything except the processor can be compromised. Therefore, BIOS, a hypervisor, and an operating system are assumed to be untrusted. SGX is based on a set of CPU instructions that can create a secure container with a dedicated address space called enclave [34]. Data inside the enclave is cryptographically signed and encrypted to ensure integrity and secrecy. Before passing to CPU for execution, integrity is verified and files are decrypted. Analogously, after finished processing, files are again encrypted and cryptographically signed. This mechanism prevents bus sniffing, tampering with memory and cold boot attacks against SGX-enabled systems [132]. We use SGX sealing mechanism [9], which takes encryption a step further by combining the encryption key with additional enclave properties. It ensures that decryption can be performed only using a platform-specific enclave unique key, called a seal key. A verification policy file is encrypted using an enclave seal key and can be persisted on a physical disk outside the enclave if needed. We assume that VMs can be created on different cloud stacks, in various physical locations, or managed by independent entities. However, once they have been attested, then they all create a multi-cloud trusted compute resource pool. In consequence, application vendors are able to deploy their solution in a virtualized and distributed infrastructure with the level of trust not available before.



(a) Virtual Trusted Platform Module (b) Virtual SGX

Fig. 4.1: Comparison of vTPM and vSGX architecture in a multi-cloud environment

Intel's SGX leverages CPU as trusted hardware on a remote computer, assuming that everything else on the platform is untrusted. CPU with necessary instructions is the only hardware requirement for SGX technology as presented in Fig. 4.1b. In opposition to vTPM presented in Fig. 4.1a., it does not require any additional components to create and maintain a chain-of-trust. This makes it a good candidate for cloud deployments.

A dedicated protected SGX memory, where enclave pages and SGX instructions are stored, is called Enclave Page Cache (EPC) and is limited in size. For bare metal deployments, it is exclusively used by an operating system, but for multi-tenant environments it is shared between all VMs. A hypervisor with SGX support allocates a requested EPC memory for a particular VM once the machine is created. The allocated memory becomes the enclave of a guest VM. This enables secure execution tied to the host hardware for the attestation software on a guest VM. A remote verifier can then trust the results of an attestation process. Based on that, the verifier evaluates an application trust level. SGX-based remote attestation process for VM is composed of two steps. The goal of the first step is to provide an attestation for an enclave before it can be used by the attestation client. For this purpose, we use a well-known mechanism based on IAS. The IAS is a cloud-based service that provides remote attestation capabilities for devices that incorporate Intel hardware security features. In our proposal we use IAS, however, other solutions are also available, as discussed in Section 4.1.1.

The second step's aim is to ensure remote attestation for a VM. For this

purpose, we propose the virtual machine remote attestation schema described in Section 4.1.3.

4.1.1 Remote attestation of enclaves

Before the enclave can be used, it should also be attested, proving to a remote entity that it is trusted. We leverage IAS that, use built-in enclaves and SGX properties to prove cryptographically that our enclave is legitimate. It also assesses security level checking firmware version that is running on our system. With the standard asymmetric cryptography, where each public key has a corresponding private key, the identity of the message originator is preserved. To protect the privacy of the user and a device, Enhanced Privacy ID (EPID) mechanism is used [109]. Due to this mechanism, a single public key can have multiple private keys associated. A verifier authenticates the originator as a member of the larger group, preserving its anonymity, as well as protecting the privacy of the user and a device.

IAS raises concerns about signer's anonymity or potential service abuse [125], however, there are alternative options available. A third-party service based on Intel's DCAP [113], or the OPERA [25] can be used. They allow building one's own attestation service and claim to address privacy concerns. They also claim to address performance bottleneck possible for large scale distributed deployments. With the improved privacy and performance DCAP and OPERA claim to have, the completeness of that solutions is either not clear or questioned [110, 125]. There is still a need to address those concerns.

When the enclave attestation is completed, we know that the remote platform runs on an Intel SGX-enabled processor with the known TCB level [1]. A secure authenticated communication channel is also established, and from that point in time, a remote server can provision secrets to the enclave in a secured way. The Virtual Machine attestation process can start now as it utilizes the attested enclave.

4.1.2 Remote attestation of virtual machines

When the enclave is created and verified by the attestation service, it can be used as a secure execution environment for the attestation client. Inside the enclave, we use built-in keys to generate a private key that will be used to encrypt data before exposing it outside the enclave. The private key is not available outside the enclave, while the corresponding public key is shared with the attestation service to enable a secure communication channel. Through this channel, an encrypted policy file can be sent to the enclave with predefined filesystem checks to be executed. When testing is completed, the result is sent back in an encrypted form to the AS.

To assess the received values, the system must have knowledge about the expected values. Those values can be captured in a trusted (and controlled) laboratory environment and then used as reference values by the AS. Based on that knowledge, an assessment of a remote VM filesystem is enabled as a trusted compute resource.

4.1.3 Proposed model

The communication diagram that we propose is presented in Fig. 4.2. It is divided into two parts corresponding to the functionalities described in the previous subsections, i.e. enclave attestation and VM attestation.

To provide a proper understanding of the description of the model and to avoid potential confusion, we introduce the following terms.

- **Attestation client** (*prover*) is a piece of software that executes the required test scenarios. It is installed in a Virtual Machine.
- **Attestation service** (*verifier*) is a piece of software that controls the attestation process and can evaluate its results. It is a verifier that can assess a VM trust level installed in a trusted and fully controlled environment.
- **Attestation process** is a sequence of steps and actions allowing the verifier to assess a trust level for the enclave or Virtual Machine.

In the enclave attestation process, IAS confirms that the enclave itself is trusted. Then the proper VM attestation can commence. At a high level, the new process starts from secret provisioning, where a key pair for encryption is generated and a public key is shared with the attestation service. An encrypted policy file is pushed to the attestation client and passed to the enclave, which can decrypt and start executing test cases. The enclave is collecting test results, and making necessary application calls for system invocations. Finally, the results are encrypted and sent by the attestation client to the attestation service that can assess on a security level.

4.1.4 Communication details of the proposed model and our contribution in this area

VM attestation consists of steps that are performed to attest the VM filesystem. The numbering used below and in Fig. 4.2 is fully consistent.

1. An untrusted attestation client calls the enclave (`ecall`) to generate the asymmetric key pair. The keys are generated by the processor by EGETKEY instruction and sealed to the current enclave or to the enclave

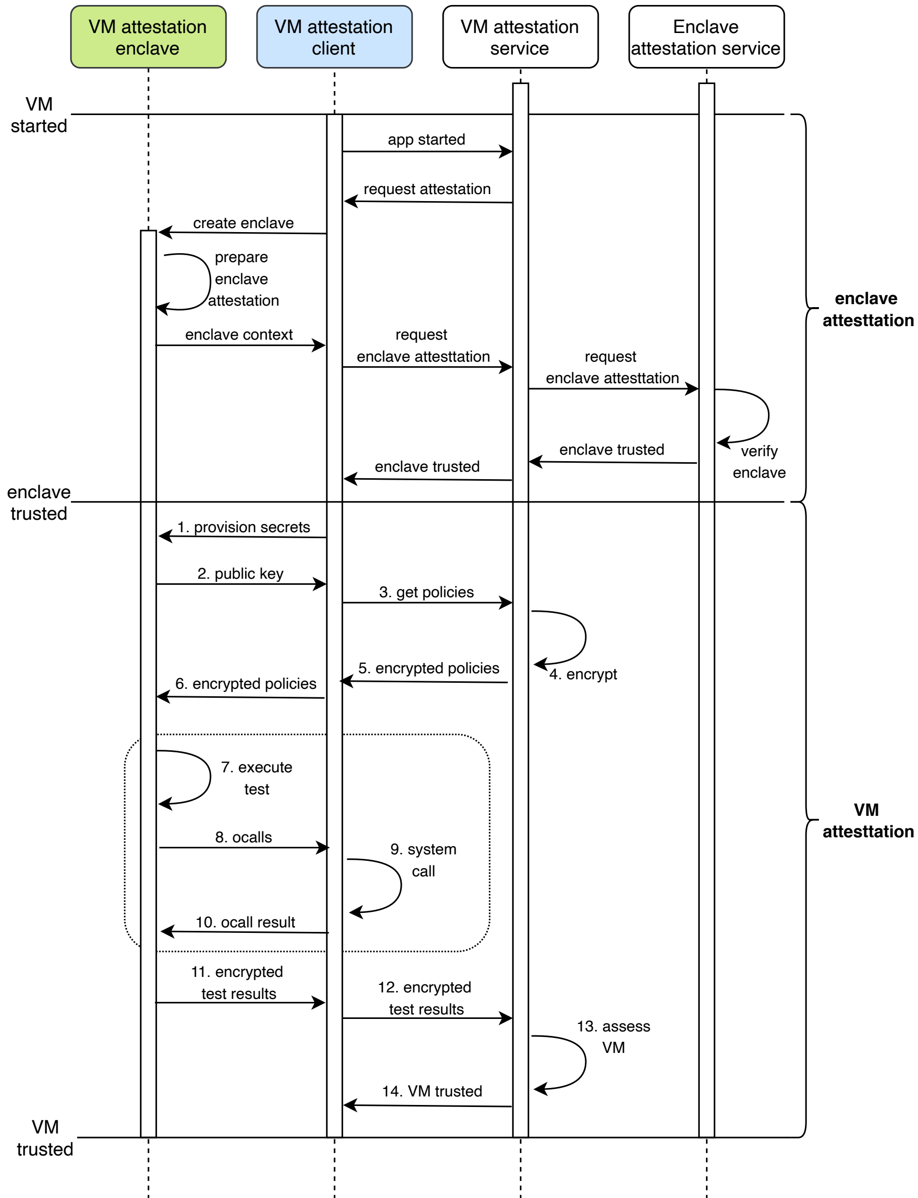


Fig. 4.2: Communication diagram for the proposed approach

signer. Therefore, only the same enclave or the enclave signed by the same identity will be able to unseal (decrypt) the data. This allows for storing encrypted data outside the enclave and for on-demand decryption inside the enclave when needed. Note that this action is started during boot time by the VM application, but it can also be triggered remotely by the attestation service on demand.

2. In a response to the previous action, the enclave returns the public key to the client.
3. The client sends the public key over the network to the AS and requests for testing policy.
4. The AS encrypts the testing policy with a public key received from the enclave.
5. Encrypted testing policy is sent back to the client.
6. The attestation client passes an encrypted policy file to the enclave that holds a private key paired with the public key.
7. The policy file is decrypted inside the enclave and the enclave starts executing the requested tests. A single test can comprise computing checksums of a list of files.
8. During computations, if there is a need to perform a system call from the enclave, the enclave has to make a call to the attestation client (ocall). This mechanism prevents the execution of any code and any action inside the enclave (e.g. IO operations, system, or external library calls). Each expected system ocall is intentionally implemented in the application.
9. The client is executing the requested system call without any additional processing.
10. The system call result is sent back to the enclave. Steps 7–10 are repeated until the testing process is completed.
11. The test results are encrypted by a private seal key and returned to the client.
12. The client sends the encrypted test results to the AS over the secured communication channel.
13. The AS decrypts results and compares them to the known appropriate values making an assessment on a VM trust level.

14. The AS makes the assessment status available for the client and other systems so that they can start using the VM as a trusted resource for computations.

Note that any network communication should use encrypted and secured channels (this fact is not depicted in Fig. 4.2). At least, TLS communication and a network firewall are required.

4.1.5 Security properties

In order to enable integrity verification of a remote Virtual Machine based on a CPU SGX instructions, we designed our system to provide the following security properties:

- **Private key protection**. Intel SGX guarantees that the key generated by a CPU is never shared outside the enclave. A new key is generated every time the enclave is created and is never persisted. This happens on every VM startup or on demand (at any point in time later when the attestation is started). In order to compromise the key, an attacker would have to find a vulnerability in either the Intel SGX architecture, or in the attestation client. Recent research demonstrated that the enclave RSA key can be reconstructed with successful side-channel attack [115] in the isolated lab environment. In a real deployment scenario, this is very challenging; however, the authors provide countermeasures that can be considered for the application development.

- **Runtime protection**. The enclave acts like a black box for the attestation client; i.e., it receives encrypted data and replies also with the encrypted data. Checksum calculation is performed in a shielded enclave runtime using encrypted memory. Only simple system calls are executed outside the enclave, minimizing opportunities for abuse to occur. When a system call is tampered, the attestation result evaluation will reveal this and the VM will not be trusted.

- **Minimal TCB**. With our model, we do not trust a cloud provider. The Trusted Computing Base is limited to Intel CPU and SGX SDK that can leverage EPC memory exposed by a hypervisor. Processor based hardware root-of-trust is inherently trusted, and provides a foundation to build security. SGX SDK executes CPU instructions in a protected runtime created for a virtual machine. The attestation client is not aware of the underlying hardware infrastructure and does not require any additional modules other than Intel CPU and SGX SDK. With this property, we can leverage various cloud models, i.e. Bare-Metal-as-a-Service/MaaS, IaaS, or system containers with a higher level of trust than before.

Table 4.1: Comparison of TPM vs. SGX based remote attestation for VMs that we propose

	Property	**vTPM**	**vSGX**
1	Availability	Poor	Limited
2	Architecture	Complex	Simple
3	Ownership	Shared	Exclusive
4	Measurements	Limited	Unlimited
5	Runtime Security	Missing	Built-in

- **Trusted Attestation Service**. In our system, the Attestation Service is the only piece of software deployed in a trusted environment owned and fully controlled by the application vendor. It means that this piece of software can also be considered trusted. Any application deployed in a cloud model is at risk due to cloud specific vulnerabilities [119], which are not a concern for the AS not deployed in a cloud.

- **Communication encryption**. Every network communication channel is TLS protected and the enclave input/output is additionally encrypted. Testing results returned by the enclave are encrypted with the enclave private key. The attestation client uses the TLS session key to further encrypt them before sending them through the network to the AS. The TLS communication is enhanced by a Client Certificate Authentication that will mutually authenticate the Attestation Client and the AS to each other. For production deployment, a TLS certificate provisioning and rotation mechanism should be ensured.

4.2 Comparison of virtual TPM and SGX based remote attestation

An application vendor who wants to verify integrity of its deployment in a cloud may consider one of the existing proposals or our SGX model. The most widespread solution that can be considered here is based on a vTPM [108]. Both solutions have limitations that one should consider, see Table 4.1.

In terms of the availability (row no. 1 in Table 4.1) of both solutions, some public cloud vendors started offering SGX for their virtual machines. They also can provide Hardware Security Module (HSM) that can be used for cryptographic operations and as secure storage, but it does not offer full TPM capabilities. The

existing TPM solutions for cloud stacks are software-based; thus, they do not guarantee the expected security. A concept of a TPM-as-a-Service [78] enables trusted computing for hardware that is not equipped with a TPM chip and increases availability. With this model, a cluster of physical TPM modules is virtualized by a cloud and exposed as a service to individual consumers. They can use TPM on demand, and migrate between platforms without a risk that the already established chain of trust to a local TCB is lost. A complex architecture is a significant reason for poor TPM availability among cloud vendors.

From the system architecture perspective (row no. 2 in Table 4.1), it is challenging for vTPM-based solution to re-establish a chain-of-trust, when VM is migrated to another host (see Fig. 4.1a). A TPM Manager must track those changes. This fact adds to the implementation complex as this component should also provide necessary availability, disaster recovery, and performance. Contrary, virtual SGX is based on processor instructions. It does not require any additional software other than SGX-enabled hypervisor.

The ownership of the solution (see row no 3 in Table 4.1) is another constraint. Our solution mitigates this issue by employing Intel's SGX, which does not require any third-party components. It benefits from CPU instructions available inside a VM, that are fully under the application's vendor control. In a vTPM model, each VM requires a dedicated vTPM instance that is connected to a physical TPM through a TPM Manager [99]. In such situations, the responsibility and process ownership is shared. This component is typically managed by the cloud vendor, implying that one must place trust in another entity's secure vTPM implementation and depend on its proper functioning.

The two solutions differ in their measurement capabilities as well, as pointed in row no. 4 in Table 4.1. For any remote attestation process, where the filesystem is verified, it is important to know where the integrity was violated. Our proposal allows us to measure every single file and send individual results to a remote system for verification. TPM, however, has a limited number of 24 PCR registers that are used to store measurement values. It uses one-way hash chaining and stores only the last value in a PCR. This means that we lose the information on what was exactly modified in a filesystem — we only know that something was modified. With this approach, and missing detailed information of the change, additional action is required to see where the integrity was violated.

Another important factor that differs vTPM solution from our proposal is a runtime security for a prover (row no. 5 in Table 4.1), i.e. a software that collects system measurements. Note, that here we discuss a runtime security for a components used for static attestation only and in the following chapters we detail on our proposal for runtime security of the entire system. vTPM does not provide a secure execution environment and is vulnerable to runtime attack without additional measures in place. A solution to protect vTPM instances by

running them inside SGX enclaves was proposed by Sun et al. [124]. The authors proposed a software based component, that emulates TPM functions which runs inside the SGX enclave. This enhances software TPM with runtime protection by storing and processing sensitive information in the isolated and encrypted memory. A physical TPM is used in this model to measure the integrity of a platform running the enclave. A similar approach, where SGX is used to enable some capabilities for a virtual TPM, is presented by Wang et al. [134]. They proposed architecture to protect runtime for vTPM instances with SGX. TPM is not employed in their model; instead, they use SGX to establish a chain-of-trust. They also identify and solve a few security challenges that exist when vTPM runtime is protected by SGX.

Several studies show how SGX can be used in virtualized infrastructures to overcome security issues in the public cloud by executing sensitive processing inside the enclave. Scone [13] can secure containers, Trusted Click [35] outsource sensitive network functions using dedicated library and use SGX to secure them. This approach faces performance challenges [133] that are addressed by processing only sensitive elements of the Virtual Network Function (VNF) inside the enclave. In opposition to these models, where code execution is delegated to the enclave, we propose to verify key components of the VM (or even an entire filesystem and runtime) to gain confidence that there is no malicious code inside. The aforementioned models also reveal the problem with the necessary application decomposition. It has to be performed during development to delegate sensitive code processing to the enclave. We propose the attestation client to be the only piece of software that has to be installed on a Virtual Machine, while other applications can remain unchanged. There are no limitations imposed on a size of the VM. Moreover, the policy can be adjusted for every execution to ensure that the measurement process is not consuming too much CPU during operations. Our system does not guarantee secure computations for sensitive data, but the attestation increasing the trust for a remote system allows us to make a decision about virtual deployment for some cases that were not considered earlier; particularly for a deployment in a public, not controlled cloud environment.

Wang et al. [134] state that SGX and TPM can be considered as complementary technologies. These authors notice that a TPM can be used to provide system integrity verification capabilities using PCR and built-in functionalities. At the same time, SGX can ensure runtime protection for the applications (like Virtual TPM) that do not ensure such security. In our model, we moved measurement capabilities to the enclave. We eliminated inconveniences caused by the problems discussed here, enabling a remote attestation for a system with a minimal SGX based TCB.

Table 4.2: Lines of code for the attestation client building blocks, rounded.

Module	Lines of code
Attestation client	800
The enclave	1100
Checksum calculator	300
In total	**2200**

4.3 Implementation details for the attestation client and the enclave

A prototype that we created for our VM attestation solution is compliant to the proposed model and presented in Fig. 4.2. It was implemented using C programming language with 2200 lines of code in total (see Table 4.2), the Intel(R) SGX SDK 2.5 for Ubuntu, and a KVM hypervisor. The latter makes EPC and SGX kernel module available for the guest VM. For the sake of comparison purposes, we do not use SGX built-in cryptography functions. Instead, we implemented md5 that can be used with and without SGX and to assess performance. This algorithm can be easily implemented without the need for additional libraries, and its vulnerabilities are not relevant in this context. We intentionally exclude the overhead necessary for result encryption and the network communication, because it is a one-time and deployment specific operation. The prototype implementation was tested on a platform running an Intel Xeon E3-1270 at 3.60 GHz with 16 GB of RAM and Ubuntu 18.04 operating system. Our single-threaded attestation client is intended for 64-bit architectures and consists of three major building blocks:

1. **The untrusted Attestation Client**. The client contains a set of routines securing the communication with the Attestation Service. We leverage OpenSSL for TLS encryption and also enforce certificate authentication with `SSL_CTX_set_verify_depth(ctx,1)` function that will enforce peer certificate verification. This allows only a certificate issued by a Certificate Authority pre-configured for the attestation client. AS presenting any other certificate will be rejected. Another major attestation client function is to execute system calls from the trusted enclave. SGX requires that every `ocall` and `ecall` is explicitly defined by the Enclave Definition Language [59], see Fig. 4.3 describing trusted and untrusted functions executed in the attestation client. Green boxes in Fig. 4.3 represent files created by *Edger* tool for trusted functions, and red boxes for untrusted.

Fig. 4.3: Trusted and untrusted functions separation by the Enclave Definition Language

This is presented in Fig. 4.3, where

The SGX Edger8 tool [59], that is shipped as a part of the Intel SDK, creates a bridge between the untrusted attestation client and the enclave, as presented in Fig. 4.4.

This mechanism adds another layer of protection for the communication between the enclave and the attestation client validating memory allocation, proper data types, direction of exchanged data structures etc. As an example, the routine given in Fig. 4.4 gets file content in the untrusted application and makes it available inside the enclave:

```
1    void ocall_fread(const char *filename,
2            unsigned char buffer[1024],
3            int *len) {...
4        //standard libc fread invocation
5        *len = fread(buffer, 1, 1024, fopen(filename, "rb"));
6    ...}
```

where:

`filename` – absolute path to the file
`buffer` – an array for max 1024 bytes
read from the file
`len` – a number of read bytes

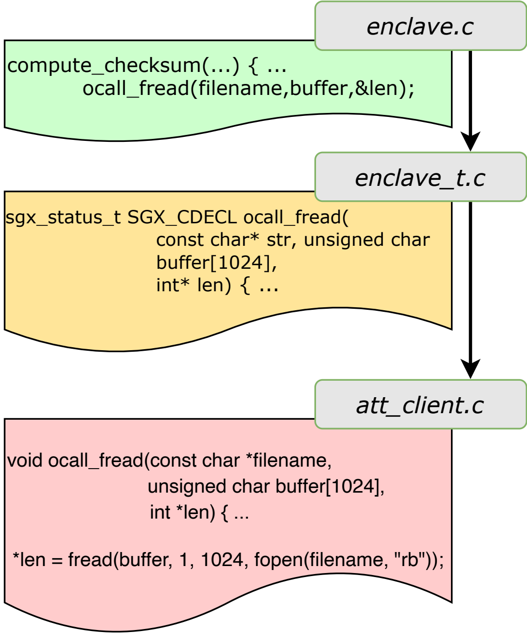


Fig. 4.4: The bridge between the enclave and the attestation client.

The function `ocall_fread` is executed in a loop until the entire file is read, automatically updating the file pointer so that subsequent `fread()` functions read subsequent file records. This is used to handle system read function from the trusted enclave for file measurement purposes. All other `ocalls` and `ecalls` are implemented in the same fashion in the attestation client and the enclave respectively.

2. **Trusted Enclave**. After it is initialized and attested, the enclave creates a key pair with `sgx_ecc256_create_key_pair()` function and shares the public key with the AS. The AS expects a policy file with a list of files to be measured or a set of commands to be executed. Optionally, the file and the partial test results can be sealed with `sgx_seal_data()` and stored externally to release some memory allocated to the enclave. The enclave leverages `ocalls` to delegate atomic command executions to the application, collects results, and repeats the process for each test. For performance evaluation, the following extra function was implemented in the enclave:

```
1     long time()
2     {
3         long int retVal;
4         if (ocall_time(&retVal) != SGX_SUCCESS)
5             abort();
6         return retVal;
7     }
```

This function returns a current system timestamp and is called before and after the execution of the test. It allows us to calculate the execution test time for the performance benchmark. The `ocall_time` is required because the enclave does not allow for any direct time functions; therefore, the execution has to be delegated outside the enclave to the attestation client. The penalty of two `ocall` executions is necessary to capture the execution time for a single test and it contributes to all test scenarios.

3. **Checksum calculator**. For our prototype, we placed this function inside both the enclave and directly in the attestation client. This is only for comparison purposes because the enclave is not in use for the **No SGX** scenario. Instead, for production usage, it is recommended to consider using trusted `libsgx_tsgxssl_crypto` which is a cryptographic SGX library based on OpenSSL 1.1.0 crypto library. It is possible because the checksum calculator is implemented only inside the enclave.

The proposed attestation client communicates with AS as shown in Fig 4.2 and described in the following chapter.

4.4 Implementation details for the attestation server

We designed our AS with the assumption that it should handle operational load. The performance of this implementation and scaling capabilities is evaluated in Sec 4.5.1. In Sec 4.5.4 we also describe potential improvements that can be considered for production usage.

For the AS we do not focus on the user interface or the preparation and distribution of tests. Instead, we proposed building blocks that play a critical role regarding the performance of the AS, as presented in Fig. 4.5. We propose 2-level cache, i.e. a database cache and a top-level cache on top of the database. Such design reduces a set of data that each time has to be swiped to search for a checksum and avoid expensive database calls. A database cache is an in-memory representation of a full data set read from a database during AS startup. A top-level cache is used to store correct measurements for frequently requested objects. When the same test scenario is executed for a number of VMs, this is a small and fixed data set. For diverse requests this set grows and one may consider Least Recently Used, or other caching strategy to keep it reasonably small. A top-level cache size increases when measured VMs are not the same, for example, in terms of operating systems and files to be checked. In a corner case, this results in unique checksums to be verified each time and the AS cannot effectively use top-level cache. The AS leverage caching and performs a number of steps to evaluate the measurements received from the VM. The numbering used in the following and in Fig. 4.5 is fully consistent.

1. An encrypted measurement file is received by the AS over a secure TLS channel.

2. A secure communication is terminated and a file is additionally decrypted to a plain text that can be processed by the AS.

3. Each line in a file contains an absolute file path and a checksum result. The lines are sequentially processed in a loop.

4. Check if the checksum exists in a top-level cache, i.e. in most frequently requested objects. A comparator block searches for a checksum for an absolute file path. If it matches the received value, then there is no action, and the execution continues with the next item.

5. For a top-level cache miss, the checksum is verified in a database cache. Due to a number of records it can store, it is slower than a top-level cache. When a checksum is found there, it is populated to a top-level cache to speed up a next request for the same object. For repetitive test scenarios, it should be

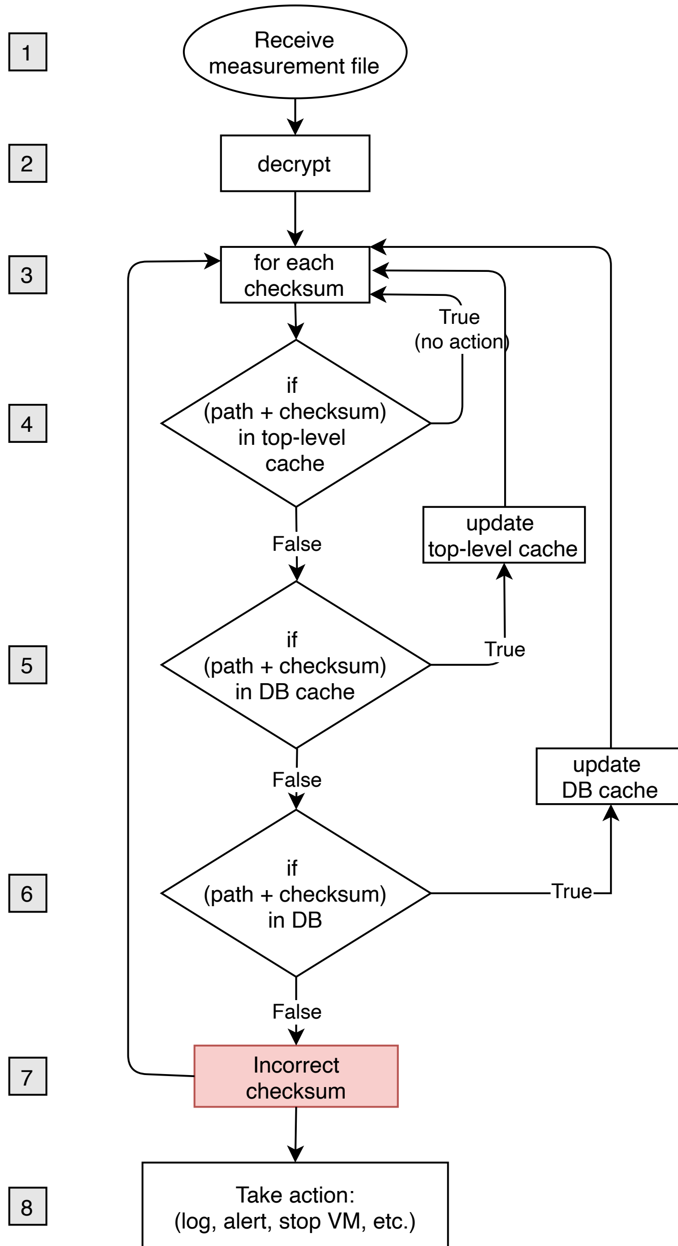


Fig. 4.5: Attestation server algorithm

used only for the first run because subsequent calls for the same checksum will result in a top-level cache hit.

6. A direct check in a database is a last resort action. This can happen when the DB cache is out-of-sync with a database caused, for example, by an update from the user interface. Such an update should trigger DB cache refresh. Cache misses may occur only until the update is completed.
7. When the checksum is not found, neither in any of the caches nor a database file integrity violation is detected.
8. There are a couple of options that can be considered when unknown checksum was found. At a minimum, this should be reported for further investigation. A severity of discovered inconsistency should be evaluated before taking ultimate action, e.g., by stopping a VM.

In the following section, we analyze a performance for a proposed algorithm. This is necessary to ensure that our AS can scale and handle expected production load.

4.5 Evaluation of the proposed system

We have examined several aspects of the proposed system that are most important for potential implementation in a real-life scenario. They are related to performance and scalability, which often determine the implementation of a given solution. At first, we measured a processing overhead generated by our solution for varying in size files. We analyzed a linear dependency between file size and a processing time that is observed for files larger then 10 kB. We calculated the formula that can be used to estimate measurement time for any file size. In addition to the client performance, we verified the AS performance and scaling capabilities critical for large scale and distributed VM deployments. Moreover, we also identified potential improvements, that may further increase processing capabilities of our solution.

We start our evaluation defining three scenarios that were used to measure a processing time for three types of files. These types represent files that vary in size, ranging from the smallest configuration files to medium-sized application binaries or libraries, and the largest archives. Measurement results were used to evaluate performance overhead when an SGX enclave is introduced for a virtual machine. It was done by computing a checksum for the selected files in the three following scenarios:

- **No SGX**. The attestation client is deployed directly on a bare metal operating system without any virtualization and does not use SGX for

computations. This is used as a reference scenario for performance comparison. In this situation, conventional memory is utilized for executing all operations instead of using a secure enclave.

- **Physical SGX**. The attestation client is deployed directly on a bare metal operating system and uses SGX for computations.
- **Virtual SGX**. This is our proposal, where the attestation client is deployed inside a virtual machine and is configured to use SGX enclaves exposed by a hypervisor for computations.

In each scenario, we perform 100 measurements for every batch, with each batch consisting of a selection of 50 files. In **Physical SGX** and **Virtual SGX** scenarios, the same steps are executed, but in the former case they are executed directly on a host and in the latter case it happens on a VM, which is already created and ready to use.

4.5.1 Performance evaluation of the attestation server

In order to evaluate the AS performance and measure its scaling capabilities, a sample configuration was deployed in a lab environment. The AS was deployed on two physical machines running an Intel Xeon E3-1270 at 3.60 GHz with 16 GB and 32 GB of RAM and Ubuntu 18.04 operating system. Both instances are located in the same network and share the same MySQL database. Each AS is running a webserver Nginx as a front-end terminating TLS connection. The request is then passed to the uWSGI socket and handled by AS. With such a design, we leverage very well performing components that provide robust connection handling, and TLS termination. They act as a secure application gateway that can automatically spin up the required number of AS threads. The AS is employed only for the core part of the algorithm, presented in Fig. 4.5. Two physical instances can handle load separately or act as a single instance hidden behind a DNS domain with two A records. Jmeter client is used as a load generator. It is deployed on separate hardware and does not affect AS performance. For test purposes, a database is preloaded with 100,000 records that are cached in the AS database cache. A single measurement report contains 1000 checksums to be verified. To verify processing and scaling capabilities, we execute a test plan 10 times in 3 different scenarios:

- single AS instance1,
- single AS instance2,
- both instances clustered together — this reflects horizontal scaling, where another machine is transparently added to increase processing capabilities.

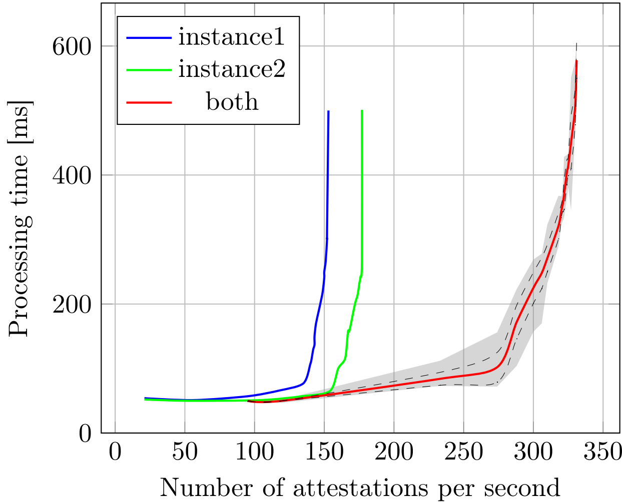


Fig. 4.6: Dependency between the number of attestations and a processing time

A dependency between the number of attestations and a processing time is presented in Fig. 4.6. Note that in one attestation step, we verify 1000 measurements captured on a virtual machine, i.e. we check the integrity of 1000 files.

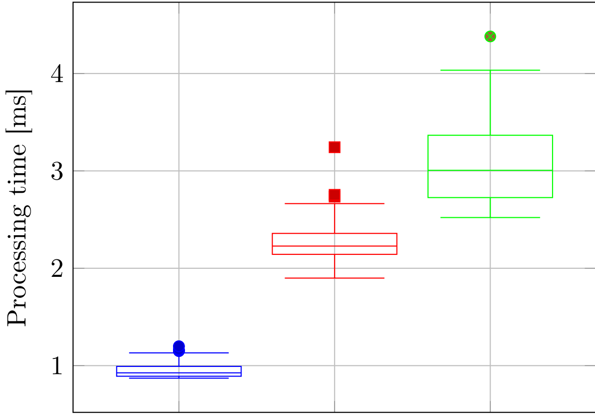
We observe that, regardless of the scenario, AS requires minimum 50 ms to evaluate the measurement result. The value is stable as the number of attestation grows, up to the point where the line changes its course. From that point on, processing time grows, which means that the machine becomes overloaded due to insufficient CPU processing power and is not able to handle more transactions per second. Instances are reaching their maximum capacity at around 150 and 175 attestations per second. A red line representing the last scenario is accompanied by a gray range of extreme values and a dashed confidence interval based on a Student's t-distribution with 95% confidence level. The red line changes its course more gently due to the statistical multiplexing gain known from queuing theory, reaching 325 attestations per second which is equal to the summarized performance levels when instances are loaded separately. This means that for our design, horizontal scaling does not introduce additional overhead and one can easily calculate the number of required machines. Assuming only a single instance (blue line), AS can evaluate 9 000 virtual machines every minute and more than 500 000 every hour. With such results, the application vendor has a lot of flexibility adjusting to attestation needs resulting from the number of owned VMs.

4.5.2 Detailed comparison

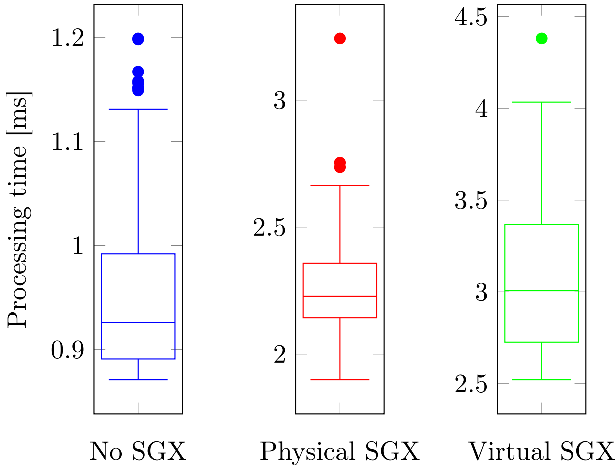
We start the evaluation of our proposal from a processing time analysis for three different file flavors. Their size has been selected intentionally, so that they represent different data types in a filesystem. The smallest files in size can reflect configuration files critical for any filesystem. Medium files are documents, system libraries, or images. Finally, the largest files represent archives, backups, or even video content. There are two graphs presented for each file type, where subfigure (a) compares the processing time for all scenarios, and the zoomed subfigure (b) shows the detailed data distribution for each of the scenarios. Note the other range on the y-axis in subfigures (b).

Fig. 4.7 presents box plots of the processing time for small 1 kB files. It can be observed in Fig. 4.7a, that the processing time median increased by 240% comparing to the **NoSGX** when SGX technology was introduced for physical deployment (**Physical SGX** scenario). This is caused by enclave data encryption and decryption as well as `ocall` executions needed to perform operations not supported by the enclave. The virtualization layer (**Virtual SGX** scenario) increased median by another 130% comparing to the reference scenario. To explain the phenomenon, we can refer to several different causes explained by Chen et al. [27]. They pointed out that for CPU-intensive tasks, hypervisor scheduling overhead and virtualized device drivers are major contributors to processing time. In cloud deployments, hardware resources like CPU or network interfaces are shared. Input/output operations are spread across multiple disks and utilized by a number of client operating systems. In such cases, the resulting overhead is expected to increase. On the other hand, optimization of VM parameters for utilization of some platform features like exclusive CPU pinning or SRiOV may reduce VM overhead. With this kind of optimization, it is possible to achieve near-native performance. The small size of the interquartile range in Fig. 4.7b for **No SGX** scenario means that hardware is able to provide stable computing capabilities when the enclave is not used and expensive context switching is avoided. The results are more distributed for **Physical SGX** and the effect is even more noticeable for **Virtual SGX**, where the spread of the data is the result of more complex processing and involvement of the virtualization layer.

A processing time for medium-size 1 MB files typical for libraries or executables and large 1 GB files representing binaries, videos, and archives are presented in Fig. 4.8, and Fig. 4.9, respectively. One must note that the order of magnitude of measured processing time, in comparison to 1 MB files has been changed in Fig. 4.9 from ms to seconds. It is observed that the median value increased for both file flavors by 100% when SGX was introduced (**Physical SGX**) comparing to the **NoSGX** reference scenario. This gives us a clue that linear dependency may occur; and thus, we performed the corresponding studies described further in this thesis in Section 4.5.3.

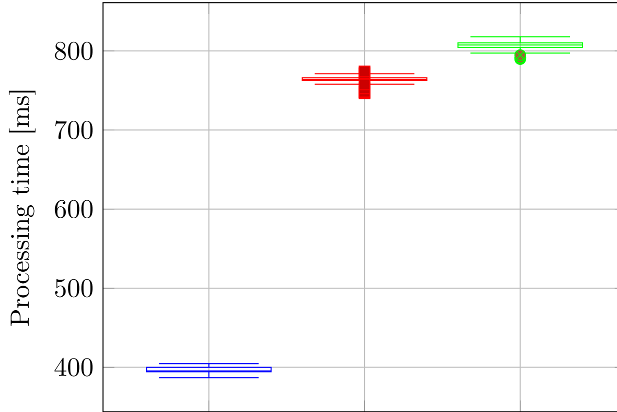


(a) General comparison

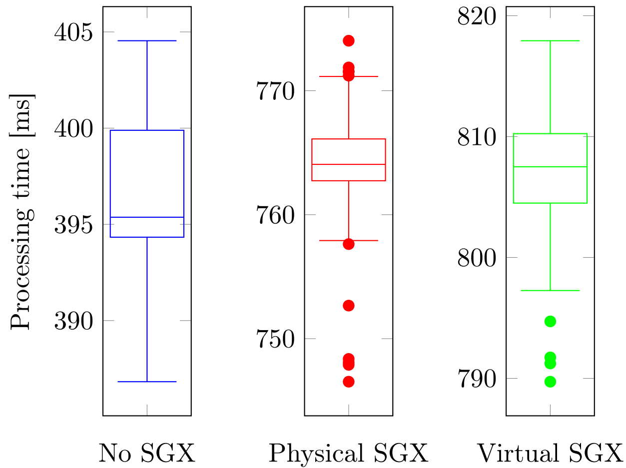


(b) Detailed distribution

Fig. 4.7: Processing time for 50 files, avg size 1 kB

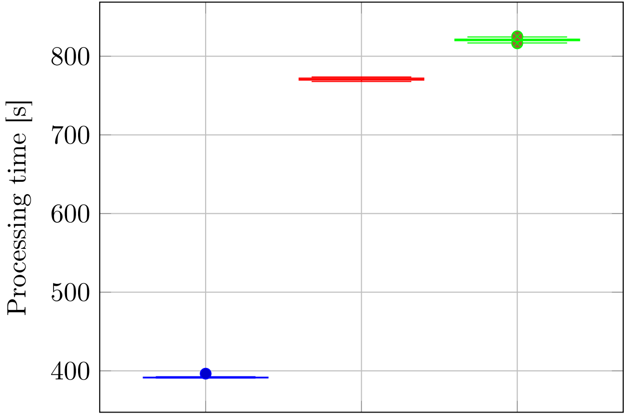


(a) General comparison

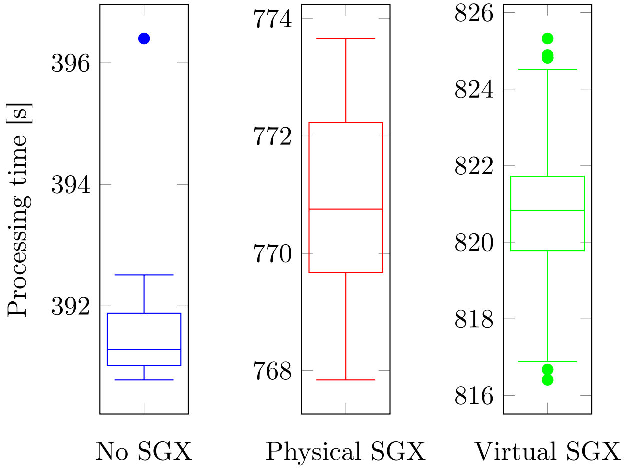


(b) Detailed distribution

Fig. 4.8: Processing time for 50 files, avg size 1 MB



(a) General comparison



(b) Detailed distribution

Fig. 4.9: Processing time for 50 files, avg size 1 GB

The introduction of the virtualization layer resulted in a further increase in processing time of 6% when the hypervisor was introduced (**Virtual SGX**). Figs. 4.8–4.9 show a significant impact on the output performance, where SGX is involved in processing. Such files need to be divided into small chunks before measuring inside the enclave. Thus, the number of `ocall` executions is significant. The cost of context switching between the attestation client and the enclave varies between 10 000 to 18 000 CPU cycles [44]. This means that for 1 GB file with 1 kB chunk size, our system needs 10^6 attestation client calls, for our 3.60 GHz CPU. This results in 3 to 5 sec. overhead for such a file.

It can be also noticed in Fig. 4.9b that upper outliers for large files are reported when a long-running test is affected by the execution of side operating system processes that compete for resources. It happens when the operating system allocates the CPU to the process with a higher priority and is observable for single-threaded attestation clients that utilize a CPU shared between multiple VMs. Such overprovisioning, typical for cloud deployments, can be mitigated by pinning CPU to VM by allocating processing power exclusively for the attestation client.

The presented results do not comprise two factors that will further contribute to processing time. An encryption is excluded because it is a one-time operation that may introduce inconsistency between the compared scenarios. The reason is that encryption is provided by different libraries for various scenarios. The enclave will use an internally generated enclave seal key and a built-in encryption engine. Similar functionalities are available in bare metal but with another implementation and cannot be benchmarked together. The second factor regards the communication between the attestation client and AS. This aspect is omitted because it varies between deployments, and may disrupt results, too. To avoid this overhead, in our prototype, the AS and the attestation client are deployed on the same physical machine and use a TCP connection to communicate. In a real deployment, there is a networking overhead that may bring significant overhead, but it is not CPU expensive; thus, it does not affect VM computing operations critical for measurement.

4.5.3 Overhead added by the execution environment

To visualize the overhead of the execution environment, a dependency between the size of files and the processing time for the three scenarios and the 50 tested files was studied. The results are presented in the log–log plot in Fig. 4.10.

The size of the smallest file in this test is 2^0 byte (1 B), and it is doubled in each iteration, reaching 2^{30} bytes (1 GB) for the last one. The first general conclusion is that introduction of SGX either in physical or virtual deployments brings additional processing overhead.

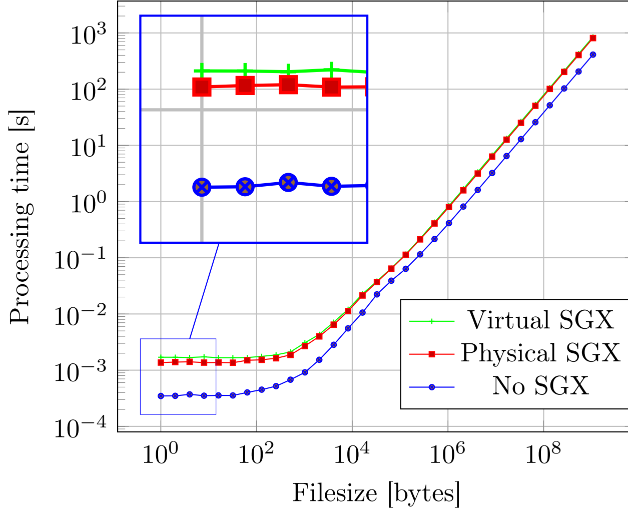


Fig. 4.10: Dependency between filesize and processing time for 50 files for Virtual SGX

This experiment shows that processing time does not increase for small files as a function of file size. It means that the measurement operations for file sizes representing configuration files are not significant contributors to the overall processing time. In this case, the execution environment overhead necessary to provide measurement capabilities, resulting from object creation, memory allocation, library calls, etc., is a major contributor. This overhead is approximately equal to 3.5×10^{-4} sec. for **No SGX**, 1.4×10^{-3} sec. for **Physical SGX** and 1.7×10^{-3} sec. for **Virtual SGX**. With increasing file sizes, measurement time starts to play a significant role, and the linear dependency[1] between file size and processing time can be observed in Fig. 4.10 starting from file sizes of 10 kB. To prove this, we calculated the Paerson's correlation coefficient ($r = 0.9999998$) and the formula that can be used to estimate measurement time (y) for any file size (x):

$$y = 8 \times 10^{-7} - 0.0336 \tag{4.1}$$

As an example, such a calculation for 1 TB file is presented in Table 4.3 together with processing times measured for smaller files. The greatest deviation of results is reported for small files measured in the most complex **Virtual SGX** scenario.

[1]The slope in the log–log plot equals 1.

Table 4.3: Processing time for a singe file (avg values)

File Size	No SGX	Physical SGX	Virtual SGX
1 kB	0.02 ms ±6.0%	0.05 ms ±6.7%	0.06 ms ±7.9%
1 MB	8.3 ms ±1.0%	16.0 ms ±1.7%	16.8 ms ±1.7%
1 GB	8.3 sec. ±0.03%	16.2 sec, ±1.8%	17.0 sec. ±1.8%
(1 TB) (est.)	(8426 sec.) (2 hr 20 min. 26 sec.)	(16 583 sec.) (4 hr 36 min. 23 s)	(17 377 sec.) (4 hr 49 min. 37 sec.)

On the other hand, the results indicate the largest stability for huge files processed in the bare metal **No SGX** scenario.

4.5.4 Potential improvements for real-life implementation

In the proposed model, there are three components that communicate over the network with each other: (a) VM attestation client using the enclave, (b) VM Attestation Server, (c) the enclave attestation service. For the enclave attestation service we assumed the existing IAS, which is production ready API maintained entirely by Intel.

The attestation client is deployed inside a VM together with other application vendor services. For a typical cloud deployment, such a client can be incorporated into the VM image used to boot an instance, or it can be installed during the bootstrapping process. It is expected that every VM should run the attestation client in the background together with other operating system processes. The attestation client does not induce any deployment cost and does not have to scale as a part of the VM.

AS is a single point of contact for all attestation clients and should be run in a trusted and fully controlled environment. For production deployment, where high availability is obligatory, one should take necessary actions to eliminate this single point of failure. The load of AS will increase as the number of attestation clients increases. The AS is a place where a bottleneck can occur if the traffic grows. Several measures can be taken to improve the performance and scalability of our solution. This includes, but is not limited to:

- **Algorithm selection**. A more efficient algorithm for file integrity check from a built-in security library can be used to improve checksum or hash calculation.

- **Multi-threading**. The attestation client may use multiple threads to process more files at the same time.
- **Code optimization**. With some engineering effort and deep analysis, it is possible to improve a client code to use optimal buffer sizes, reduce complexity, and the attestation client calls.
- **VM tuning**. Our model is CPU expensive and one should consider that fact by allocating more CPUs or exclusively pinning CPU to a VM on creation
- **Exitless system calls** [75]. For each system call, the process must exit the enclave increasing performance overhead [92]. Asynchronous call invocations could reduce this overhead.
- **Reliable database backend** for the AS data persistence storing test scenarios, results, attestation clients information, etc.
- **Multiple Attestation Server instances.** At least one more standby instance is required that can take over operations when the primary instance is down. Additional instances should be deployed to handle more clients.
- **Load balancing.** Depending on the expected load generated by distributed attestation clients, traffic to the attestation server can be load balanced. One can take advantage of virtual IP assignment inside a data center for local load balancing and increase the number of backend hosts when traffic grows. Additionally, traffic can be distributed globally with multiple A records in DNS pointing to different attestation server deployments in different physical locations.

One must note that the exact architecture depends on the requirements, and they may vary for every deployment. The highlighted engineering improvements do not change the proposed functionality. They can reduce execution time in practical deployments of our solution.

4.6 Security considerations

Apart from the traditional adversaries, such as malware, hackers, spammers, etc., there are also cloud specific threats for the cloud deployments. These threats include malicious cloud providers, malicious tenants, or insiders. With the Intel SGX threat model, only the enclave is trusted, and we assume that the adversary can access virtual machine, hypervisor, cloud infrastructure, and underlying hardware components. The system software is considered untrusted, but the SGX design prevents it from reading the enclave memory and data. It means

that any processing performed inside the enclave is secure, and any untrusted attestation client call can be compromised by malware or rootkits. Such a call is required when the enclave is looking for information not available directly for the enclave. If that happens, the proposed system detects incorrect attestation results and denotes a particular virtual machine as compromised. Hypothetically, the adversary can pass to the enclave expected information, but in that case the adversary's actions must leave some tracks in the filesystem. The entire system attestation check reveals this and raises an alert for this VM. It is important to note that the adversary is not aware of the testing policy and expected results. We can take advantage of that fact and also evaluate the measurement time or any command execution, which can give us a full picture of what is happening inside our system. As an example for a given system, the measurement time of file X should take a time: $t_{\min} < t_X < t_{\max}$. For a single-threaded attestation client, measuring a set of files will also generate a vector of individual measurement time values: $v = t_A, t_B, t_C, \ldots, t_N$. This information can be used as a result fingerprint validated by the AS. If a measured time is outside of a predefined time range, we can infer that the value was injected. We do not consider any Denial of Service attacks, including network layer or resource exhaustion. The VM is trusted only for fully functional end-to-end operations. We also do not implement any mechanism to protect against side-channel attacks, since the solutions described in [79, 91, 115] can be adopted to mitigate them.

We have shown that our system can be used successfully to test VMs in a distributed cloud environment running SGX. However, such deployments are very complex and we do not always have knowledge about the underlying infrastructure. We should be cautious when deploying the solution and remember about the specifics of the cloud. Otherwise, we may expose it to additional security risks. For example, apart from basic lifecycle operations, the cloud software stack should also provide more complex functionalities, e.g. VM migration or scaling. For a typical SGX based application that was migrated, the enclave uses a new processor that creates a different pair of keys. This means that you will not be able to decrypt data encrypted using keys generated by the processor of the previous virtual machine. In such cases, a secure method for key exchange and encrypted enclave data migration between old and new platforms should be available. The suggested extensions [54, 95, 96] are promising but not mature. Moreover, they require additional building blocks. This increases complexity and does not guarantee confidentiality, integrity, and availability of the migrated data, therefore the VM live migration cannot be considered secure.

In our model, we do not migrate the enclave data; therefore, we support VM migration. We can store data outside the enclave only when the attestation process is running. Moreover, always when the VM is started, we assume that all data is lost and that the VM is not trusted. This strong assumption means that

the VM must go through the attestation process before it can handle operational traffic. Due to this property, the proposed solution supports any cloud operation that requires using another processor, including VM migration.

It should be noted that static attestation validates only static files as depicted in Fig. 2.2, and one must trust that the application using them will behave properly. This assumption is not valid for control flow integrity attacks in which an unmodified binary is misused. Additionally, the process does not assess the integrity in real-time and can report violation some time after the violation occurred. This puts the reliability of the evaluation (step 3) in question and raises the demand for a solution that can address missing area. In Chapter 5 we fill this gap extending our proposal with runtime attestation capabilities.

5 Runtime attestation

Runtime attestation is a technique for verifying the integrity of a system while it is running; see Section 2.4 for more details. It provides a way to detect and respond to runtime attacks in real-time, enabling organizations to take proactive measures to protect their systems. This technique involves creating a Trusted Execution Environment [83, 84, 110] that can monitor the system's runtime behavior and detect any anomalies or unexpected changes.

The TEE is created using hardware-based security mechanisms such as TPM or secure enclaves. These mechanisms provide a secure environment that is isolated from the rest of the system and can perform secure measurements of the system's runtime state. These measurements are then compared to a predefined set of measurements, called a policy, to determine whether the system is behaving as expected.

If the TEE detects any anomalies or unexpected changes, it can trigger a response, such as shutting down the system, alerting security personnel, or taking other remedial actions. This approach can help prevent runtime attacks and minimize their impact by detecting them early and responding quickly.

Remote attestation for control flow is a critical process to ensure the integrity of application runtime. We propose a solution that can effectively enforce runtime integrity and prevent control-flow hijacking. Additionally, the solution can prevent from executing not protected applications.

This chapter is based on two of our publications [73, 74], which provide comprehensive insights and contributions to the existing body of knowledge. In next section we evaluate hardware support for Control-Flow Integrity based on first available implementation. This is to confirm that CET will introduce fully acceptable overhead when running on a bare metal. With positive results, we apply this technology to our runtime attestation solution for virtual deployments

in the following sections. It complements presented in Chapter 4 model with runtime attestation capabilities increasing the application security.

5.1 Evaluation of the Control Flow Integrity Enforcement

For several years, flow integrity problems have been researched by many projects. Proposed solutions have limitations, as described in Section 2.1; therefore, they are not adopted by the industry on a required scale. The efficiency of emerging concepts in this area, such as Intel CET, are likely to be questioned due to a poor historical experience with other similar solutions. Intel CET is claimed to generate much lower performance overhead than software-based approaches. Thus, it is necessary to thoroughly verify the performance impact and possible applications for this technology for virtual deployments that recently grow in scale significantly. The assumption of our test scenarios was to measure that overhead in the following scenarios.

- **No CET**. The test program is running in a CET-enabled operating system and libraries, but it is not compiled with the required option enforcing CET.
- **CET-enabled**. The test program is running in a CET-enabled operating system and libraries; it is properly compiled to enforce CET.

Different types of software workloads have different requirements for resources such as memory, CPU, network, or disk. Therefore, any hardware-based technology that relies solely on a CPU cannot be easily verified for any software installed in the operating system. This stems from the fact that the majority of processes also require other resources than pure CPU. The resources are typically limited; thus, they can impact performance results when used. We discuss our testing approach in the following subsection.

5.1.1 Methodology of the tests

A typical computer system consists of many programs of varying complexity. Depending on the purpose, one can find a huge database, a webserver, AI engine, or, on the other hand, simple and atomic processes dedicated for particular tasks (file viewer, system utilities, image compressor, etc.). We do not consider the former, since a substantial contributor to the processing time is usually not a CPU, but other system resources. In such a case, the overhead of the evaluated technology is negligible. Instead, we focus on small and atomic programs that show heavy CPU utilization when loaded. Each program tested by us was compiled from sources with two flavors, i.e., with and without hardware support for a

control-flow validation. Additionally, we implemented two simple test programs (*test-subroutines.c* and *test-math.c*) that can generate different types of CPU loads. Implemented programs use different CPU parts to generate similar processing time workloads. Moreover, *test-math.c* intentionally does not invoke any other function. The following listings show a pseudo-code for both of them.

test-subroutines.c:

```
1  main() {for(i=0; i<ITER; i++) f1()};
2  f1()   { return f2(); }
3  f2()   { return f3(); }
4  f3()   { return f4(); }
5  f4()   { return f5(); }
6  f5()   { return f6(); }
7  f6()   { return f7(); }
8  f7()   { return f8(); }
9  f8()   { return f9(); }
10 f9()   { return f10(); }
11 f10()  { return 1; }
```

test-math.c:

```
1  main() {for(i= 0;i <ITER;i++){}};
```

Very short execution times can be measured by available system libraries, but that would involve changes in the application source code to capture time data. To avoid such code instrumentation, we adjust the computation problem, as detailed in a workload column in Table 5.1. In such case, the execution takes more time and can be measured with any available system tools. The time taken to complete the task without CET protection in place and with CET-enabled binary is captured in **No CET** and **CET-enabled** columns respectively. We calculate the processing overhead for measured execution times and use this value to sort all programs in ascending order. Additionally, we show the impact on the binary size when it was compiled with hardware control-flow enforcement. This is caused by additional `ENDBRANCH` instructions automatically added by the compiler, as presented in the last column of Table 5.1. Such instruction is used to inform the processor that it is about to transfer control to a trusted code block. Note that even for the simplest program with only one main function, additional `ENDBRANCH` instructions are present in the binary of another linked system library.

Before the tests there run, a correct system behavior was validated with a simple ROP exploit that abuses the program behavior and calls the system standard library function. The exploit leverage the fact that the C *gets* function suffers from buffer overflow because it does not check the length of the consumed string with the allocated array. Therefore, the attacker can construct the arbitrary string, that overwrites a memory with a sequence of malicious instructions. When executed, they invoke a system shell from an already loaded shared library, opening doors for unwanted actions. With the CET-enabled system, a kernel trap is generated

Table 5.1: Test programs execution details with median processing times for a bare metal.

Name	Workload	No CET	CET-enabled	Processing overhead	Binary size increase	Number of `ENDBRANCH` instructions
test-math	arithmetic addition repeated 2.1×10^9 times	3.94 s ±0.4%	3.95 s ±0.4%	0.20%	0%	9
sha256sum	1 GB file `sha256sum`	5.68 s ±2.4%	5.69 s ±2.7%	0.18%	0.09%	257
shred	1 MB file overwrite 3000 times	4.41 s ±0.4%	4.42 s ±0.4%	0.23%	0.08%	333
gzip	1 GB file `gzip` compression	4.77 s ±7.8%	4.84 s ±9.3%	1.47%	0.09%	154
awk	word count for 35 MB file	4.23 s ±4.2%	4.37 s ±2.5%	3.31%	1.02%	1332
convert	convert 3 MB `png` to jpg with auto-levels	3.61 s ±0.8%	3.77 s ±0.9%	4.43%	0.40%	46
openssl	`openssl aes-256-cbc` encryption, 10^7 iterations	4.82 s ±0.4%	5.40 s ±0.9%	12.03%	2.21%	3547
test-subroutines	5×10^8 subroutine calls w/o body	4.01 s ±0.5%	4.65 s ±0.2%	16.00%	0%	19

and the process is terminated. With the setup prepared this way, we capture the numbers as described below.

5.1.2 Results

Our main goal was to measure performance overhead generated when the hardware protection for control-flow is enabled. The median of a processing time captured in the **No CET** and **CET-enabled** scenarios for 100 test repetitions is shown in Table 5.1. More detailed distribution is presented in Fig 5.1.

For each of the programs examined, we scaled a workload to avoid code instrumentation and present results in the same graph with the same y axis scale. As an example, for the `gzip` test, we used 1 GB file size, and for `test-math` we invoked a simple integer addition 2.1×10^9 times, as detailed in the **workload** column of the table.

The results received from *test-subroutines.c* demonstrate the 16% overhead introduced by Intel CET. This occurs because the program solely executes a subroutine call, requiring an additional invocation of the `ENDBRANCH` instruction in this technology. For this type of workload and a large artificial number of repetitions, the overhead is noticeable. On the other hand, the second (*test-math.c*) program shows a small (0.2%) overhead despite a large number of iterations. In this case, the arithmetic operations do not require any additional function invocations. CPU uses its arithmetic unit to carry out addition operation and does not have to validate `ENDBRANCH` targets. It is also seen in a graph that for these two types of tests the interquartile range is small and we do not observe visible outliers, regardless of whether CET is used or not. This means that nothing impacted the CPU while testing and the received data has little variability. With the selection

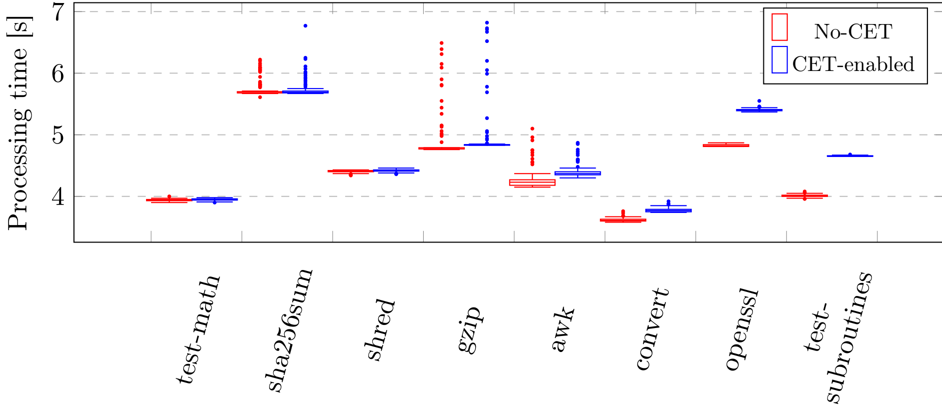


Fig. 5.1: Processing time overhead

of such test programs, we emphasize the importance of benchmarks that are used to validate the performance impact of hardware technology like CET. We consider the measured 16% performance overhead as a maximum value that could be received in special conditions, but will not be experienced in real-life applications, where one does not have to perform so many iterations. More realistic results are presented for selected programs, where the workload shows a real use case. We observe for them an overhead varying in the range 0.2-12%. A small processing overhead is noticed for programs that require resources other than only CPU (e.g., disk access). For such programs, we observe several upper outliers. This means that I/O operations contribute to the result because they do not present close and constant response times. The increase of processing overhead for such programs that require more different types of processing like `awk word count` or `convert`. This means that they require more transitions between functions of the program to complete the task, which translates into the number of `ENDBRANCH` targets to be validated.

Our tests for a bare metal, based on first available implementation, show promising results for hardware support for CFI. We also recognized the importance of a benchmark that should represent realistic use cases instead of pure mathematical problem to measure. We use this technology as a foundation for a runtime attestation solution described in the following section.

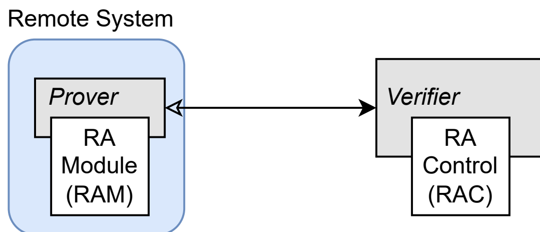


Fig. 5.2: Runtime attestation extension for prover and verifier

5.2 Solution details

We enhanced our existing solution for static attestation presented in Chapter 4 with two Runtime Attestation components (RA Module and RA Control), as presented in Fig. 5.2. A prover and a verifier are components introduced in Section 4.1.3 that we extend.

For our runtime attestation modules, we use some important properties of the existing solution, such as secured communication between the prover and verifier, secured encryption key storage for the prover or hardware root-of-trust. On top of that, we present two major building blocks for our proposal:

1. **Runtime Attestation Module (RAM)** With existing static attestation, a prover based on a received list of files from a verifier collects file checksums. The new RAM module for a prover verifies if a file is compiled with hardware support for control-flow enforcement. This information is included as a flag in a report with measurements send to verifier. Additionally, based on information received from a verifier it can block any unprotected software before execution. For any protected file that is being executed, a control-flow is enforced by a hardware. The execution is stopped when a violation is found.

2. **Runtime Attestation Control (RAC)** This module extends verifier capabilities with runtime attestation functions. A report received from prover contains file measurements and also a flag created by RAM. A RAC module evaluates entire system runtime protection, e.g. by checking how many files are protected. This information is then made available for the system owner, who can choose to leave default behavior or use strict protection mode. With default behavior, unprotected files can be executed. When a strict mode is used, RAM will block any unprotected file before execution.

With such a design, our system can effectively force runtime protection, leaving

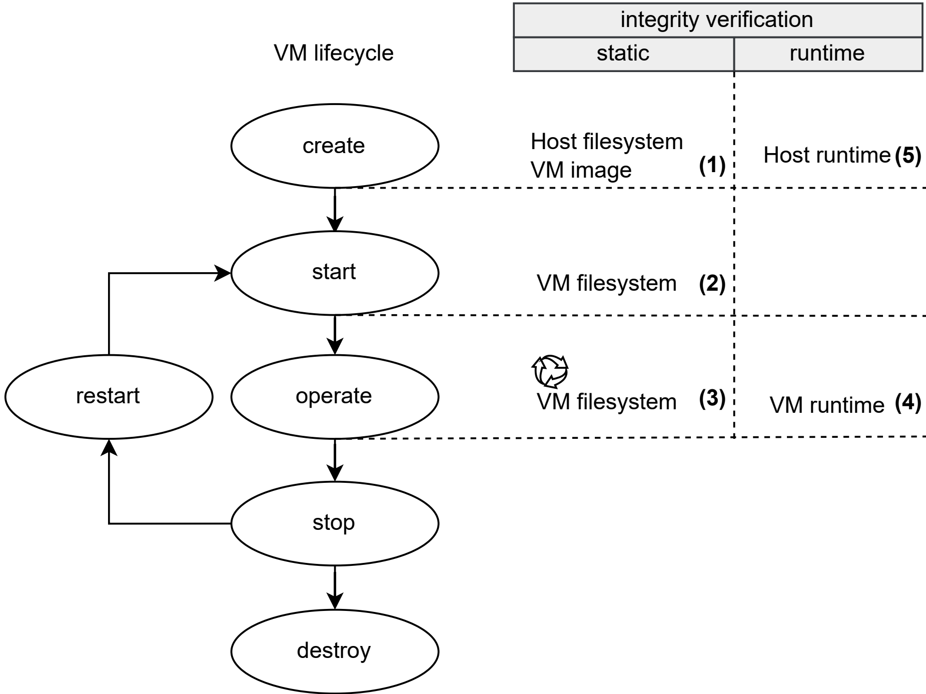


Fig. 5.3: Integrity checks through VM lifecycle

no risk that runtime protected software can exist in the same system together with any other executables or libraries. We apply our solution for a virtual infrastructure, protecting virtual machine resources throughout their lifecycle.

5.3 Static and runtime attestation through the entire VM lifecycle

Through the provisioning of VMs and their entire lifecycle, the integrity of various components can be ensured by our proposal, as presented in Fig. 5.3 and described below.

(1) An integrity for a host filesystem can be validated to make sure that the operating system and a hypervisor are not modified. In addition to that, an image used to bootstrap a VM can also be verified as it is available for a

host filesystem and copied to a compute node. In the IaaS environment, a cloud owner providing trusted compute resources should ensure this step.

(2) During startup, a VM filesystem is measured, i.e. new checksums of static files are captured to make sure it is running trusted and unmodified software. An application vendor is responsible for this step.

(3) The previous step is repeated periodically for static resources. This process validates if a software on a VM was not modified.

(4) In addition to periodic static integrity verification, a runtime is checked against malicious actions. Runtime integrity must be checked locally during program execution on the virtual machine to enforce real-time protection and deny unauthorized actions. Immediate blocking of any malicious action during code execution, without any delays caused by remote system involvement, is crucial. Our extension to static verification method, described in Section 5.2, can effectively block unprotected applications and prevent before execution when a control-flow is violated. A proposed solution for runtime integrity attestation complements the static verification method to ensure complex integrity for virtual deployments. Our solution is fully owned by the application vendor.

(5) Host runtime integrity is outside of the application vendor control. A cloud provider can use a mechanism that we propose to enforce runtime integrity and increase security for any software running on the platform.

A remote attestation process for static cloud deployment resources can be executed in three steps (numbered (1)–(3) in Fig. 5.3). The integrity check for static resources starts from a host, where a VM is to be deployed; then, a VM is validated at a startup, and finally it can be periodically checked while operating.

We propose the architecture presented in Fig. 5.4 to ensure remote application integrity attestation, based on hardware capabilities that can be easily exposed by a hypervisor to a VM. The numbers in Fig. 5.4 are consistent with those in Fig. 5.3. In (1) we additionally leverage TPM based measured boot to ensure the integrity of critical host components, and due to its limitations, we did not employ TPM for the cloud. By integrating two of our solutions, one mechanism provides described in steps (1)–(3) remote attestation for static resources presented in Chapter 4. The other mechanism enforces runtime protection in steps (4) and (5), which is the main subject of this chapter.

As presented in Fig. 5.3, we recognize five areas in which integrity should be verified on a virtual infrastructure. In the following sections, we describe how we apply our solution to these areas.

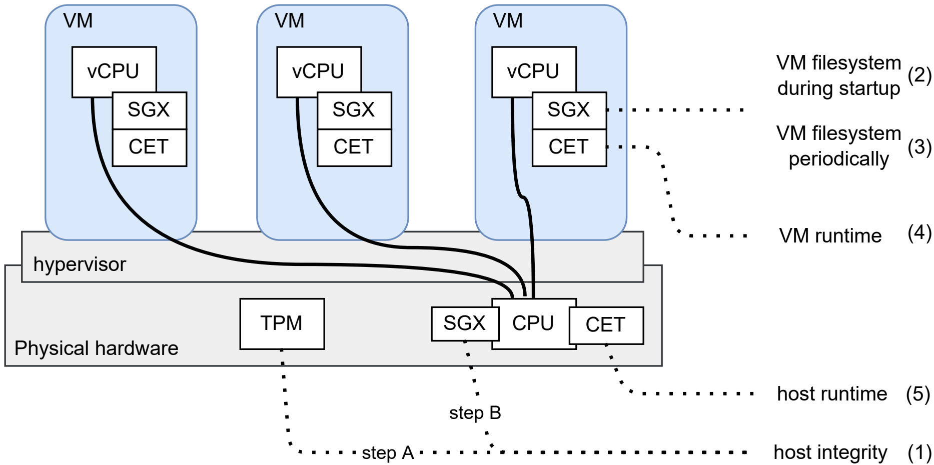


Fig. 5.4: **SGX+CET** attestation for virtual machines

The numbering is consistent with Fig. 5.3

5.3.1 Static host filesystem integrity

We enhance the measured boot based on TPM with a remote attestation based on SGX. With such a design, we extend the integrity verification capabilities to any static file located on a host machine (including VM images), as detailed in Fig. 5.5 and increase a security level for the host [51].

The process consists of two steps. The goal of step A is a measured boot, where all critical host components are verified at a startup. The first piece of code that executes on a startup is called Core Root of Trust for Measurement (CRTM). It is implicitly trusted to build a *chain-of-trust* measuring and running other components of the system, such as BIOS, firmware, bootloader, or OS. Measures are saved in PCR registers and can be used to assess key components of the host. This step is executed once, at boot-time. Step B extends the measurement capabilities to individual files within a host operating system and can be triggered on demand. We use SGX remote attestation for at least two reasons. Firstly, it allows one to measure any number of files, thus keeping track of integrity changes. Both, a platform owner and a VM owner gain insight in the integrity state of any file in a filesystem their own. Second, it establishes a chain-of-trust directly with a processor. This is a huge advantage in a cloud environment, since it does not introduce extra complexity. The information collected in this step can be exposed by a platform owner in a cloud user interface or via API. Then, a platform

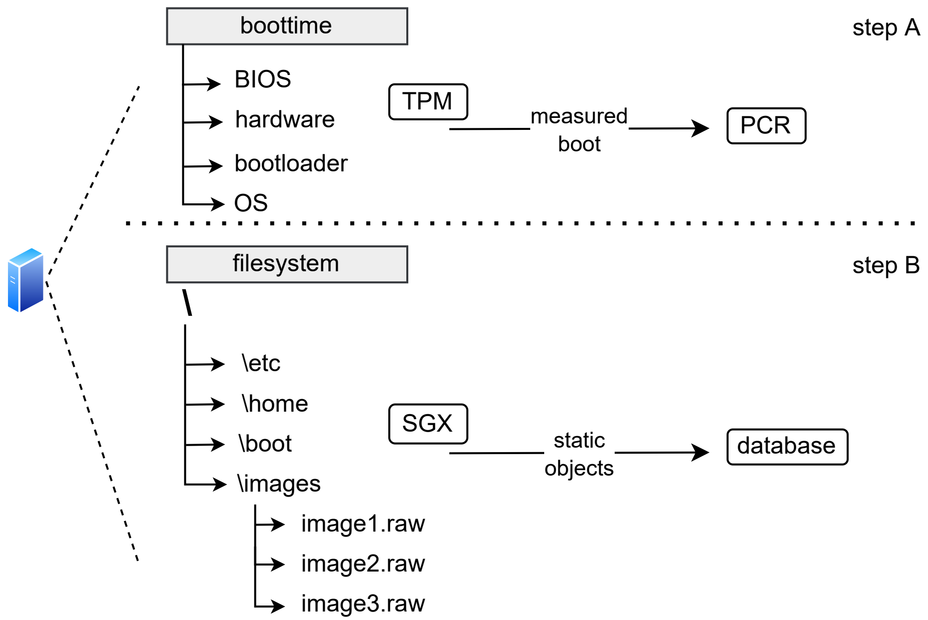


Fig. 5.5: Enhanced static host integrity validation

user can immediately verify if a source image for a VM was not modified before deploying a workload.

5.3.2 VM static integrity at a boot time

This is covered in Section 4.1, therefore, we do not discuss it here.

5.3.3 VM periodical static integrity

This is covered in Section 4.1, therefore, we do not discuss it here.

5.3.4 VM runtime integrity

We propose RAM, as an extension of the verifier, to collect information whether a software is compiled to use CET and enforce runtime protection. This enforcement is done in two ways: RAM blocks the execution of unprotected applications, and the CET instruction set prevents execution when a control-flow integrity is corrupted. To use CET inside a virtual machine, an operating system, libraries, and application must be compiled with the necessary flags. Modern operating systems and compilers already enforce this, but still a majority of software has to be recompiled.

5.3.5 Host runtime integrity

This step complements step (1) to provide static and runtime integrity protection for the host. A host, which is a part of a trusted compute pool, provides a software stack for cloud deployments. Depending on a function, it can act as a *controller* with cloud management components, or a *compute* that runs hypervisor for virtual machines. Our runtime attestation components and CET that is deployed on a host, allows a cloud owner (as opposed to a VM owner) to protect operating system and cloud software stack against runtime misuse. Because of that, a cloud user gets more trust in the environment that vm runs on.

5.4 Evaluation of our proposal

In our solution, we propose runtime attestation extensions to a prover and a verifier. The performance of a prover, a verifier and SGX operations were already evaluated in Chapter 4. Therefore, in this chapter, we focus on a performance for runtime protection that can impact operations, i.e., hardware control-flow enforcement. Before being applied to advanced use cases, each new technology has to be carefully verified. This implies functional and performance verification in a

controlled environment with a basic setup. After completion, more components should be introduced so that the environment is close to the expected configuration. In the case of a cloud infrastructure, one of the useful tests can be related to a bare metal deployment. The reason behind this is that the technology is initially integrated natively into the CPU and later becomes fully accessible within the hypervisor for a VM. We are unaware of any existing results in this area. Therefore, we measure the performance overhead in the four scenarios that describe various configurations related to CET. The test scenarios are visualized in Fig. 5.6 that shows only components involved in processing. Each subfigure is a part of the overall architecture; see Fig. 5.4, which contains all hardware building blocks for our solution. We define four scenarios in the following ways:

- CET-protected host without virtualization.

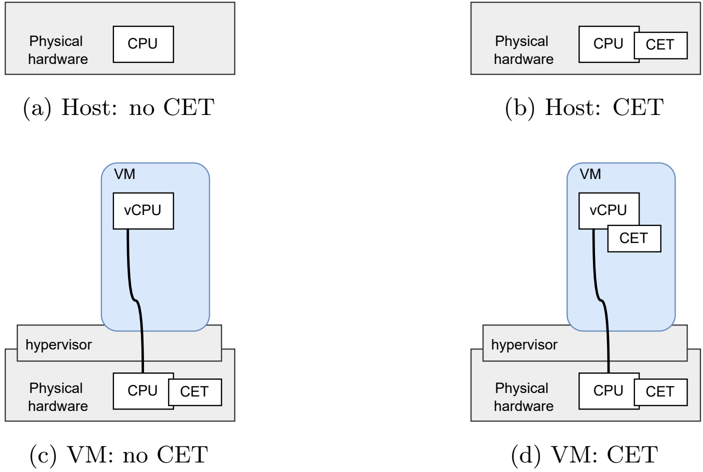
  Host: no CET. The first reference scenario assumes no virtualization. Only a host operating system is CET enabled, while a test program is compiled without the required option that enforces CET. This case will become more and more common due to the fact that modern operating systems are automatically compiled with CET support. However, available CET capability is often not used by the application owner.

  Host: CET. The test program running on a host is compiled to enforce CET. This is a recommended setup for applications running on bare metal. Application vendors can take advantage of the capabilities of the platform and leverage control-flow protection in their applications. This configuration reflects *Host runtime*, marked as (5) in Fig. 5.3

- CET-protected virtual machine running inside a CET-protected host

  VM: no CET. In this case, a virtual machine is deployed on a host that supports CET. Although, with a proper hypervisor, CET technology is also available for a virtual machine, the test program inside a virtual machine does not enforce CET. With the spread of technology, it is expected that CET will be exposed by more and more hypervisors in the industry.

  VM: CET. In this case, a virtual machine is deployed on a host that supports CET. Additionally, the test program inside a VM is compiled with the necessary flag that enforces CET. This is a recommended scenario for cloud deployments where runtime security plays a critical role. This configuration is shown as *VM runtime* in Fig. 5.3. Accompanied by the static attestation, this is a subject of our complementary solution, which we propose in this dissertation.



(a) Host: no CET (b) Host: CET

(c) VM: no CET (d) VM: CET

Fig. 5.6: Test scenarios architecture

5.4.1 Methodology of the tests

Any hardware-based technology that is part of a modern CPU cannot be easily verified. This stems from the fact that the majority of complex systems require also other resources (such as memory, network, or disk) than pure CPU. Moreover, a demand for hardware resources is not consistent; and thus varies between different types of workload. To minimize that impact, we focus on atomic programs that show fully loaded CPU when executed and serve as building blocks for advanced systems. Additionally, we use two simple test programs (*test-subroutines.c* and *test-addition.c*) described in Chapter 5 that can generate different types of CPU load (function invocations vs arithmetic operations) and do not invoke any libraries when executed.

The selected programs used for the tests are listed in Table 5.2. They represent different types of CPU load, for example: mathematical functions, file-based computations, data processing, or security-related operations (as detailed in the 'Description' column). For each test, a CPU was fully loaded. The performance of each program was measured for three workloads whose size increases linearly. A workload for every program is different; therefore, we detail the subject of scaling in the last column. Each program presented in Table 5.2 was compiled from sources with two flavors, that is, with and without hardware CPU support for control-flow validation (depending on the scenario). Each test was repeated 100 times to obtain statistically credible results, which we analyze further.

Table 5.2: Workload size for test programs

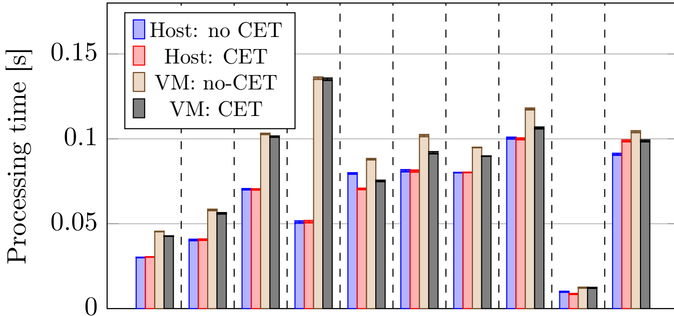
Name	Description	Workload size			What is scaled
		small	medium	large	
`test-addition`	arithmetic addition	$2{\times}10^7$	$2{\times}10^8$	$2{\times}10^9$	no. of iterations
awk	find most popular words	160 kB	1.6 MB	16 MB	file size
`zgrep`	search in a compressed file	20 kB	200 kB	2 MB	file size
convert	change of a file format	500 kB	5 MB	50 MB	image size
test-subroutine	recursive function invocations	10^6	10^7	10^8	exponent
filehash	calculate a checksum	20 MB	200 MB	2 GB	file size
encrypt	file encryption	100 MB	1 GB	10 GB	file size
bc	calculate the power of 2	$2^{2\times10^4}$	$2^{2\times10^5}$	$2^{2\times10^6}$	exponent
grep	search for string patterns	160 MB	1.6 GB	16 GB	file size
gzip	file compression	10 MB	100 MB	1 GB	file size

5.4.2 Results

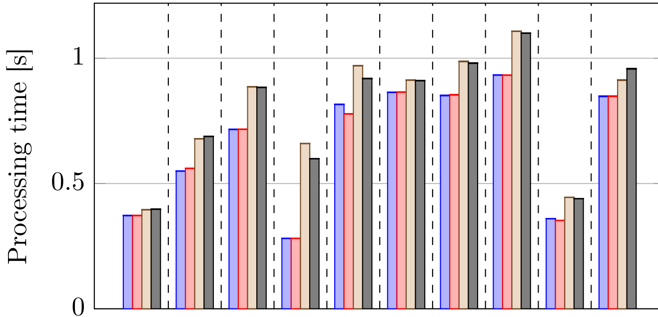
Our main goal was to measure performance overhead of the hardware-based control-flow protection for our runtime attestation solution. We apply runtime attestation for cloud deployments, a scenario described as **VM: CET** and presented in Fig. 5.6d. Unlike the existing solutions discussed in Section 3.2, which are software-based or hardware-assisted, our proposal does not involve any processing in a user space. This means that the overhead is not related to the processing of extra code; instead, it is entirely tied to the processor. We measure control-flow enforcement overhead and the virtualization overhead. Both contribute to the overall processing time presented in Fig. 5.7. Each subgraph presents a different size of workload, as detailed in Table 5.2. The test programs are placed on a horizontal axis, whereas the processing times for them are presented on a vertical axis.

confidence interval based on a Student's t-distribution with 95% confidence level.

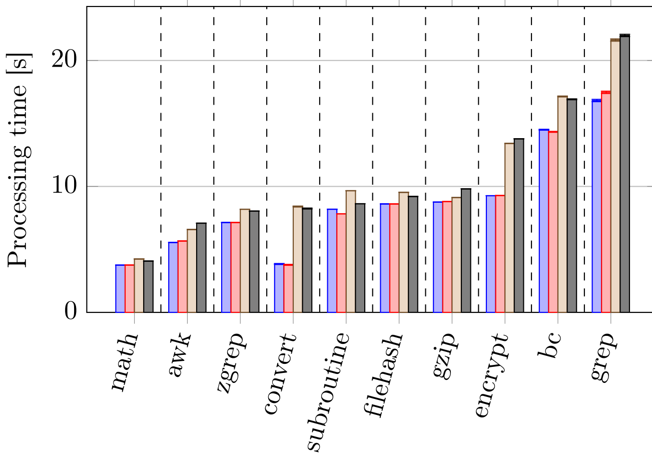
Each bar in the graph represents a test scenario as shown in Fig. 5.6 and is color coded according to the legend. The bars are grouped into four for each test program that is evaluated. For each bar, we notice a relatively narrow confidence interval based on a Student's t-distribution with 95% confidence level, barely visible at the top of the bar. For this reason, we do not show confidence intervals on subsequent graphs. The narrow values imply high stability of the measurements captured from atomic programs. Note that there is no point in comparing a processing time for totally different programs. For example, a nature of processing time for a mathematical problem related to `addition` is different



(a) small workload



(b) medium workload



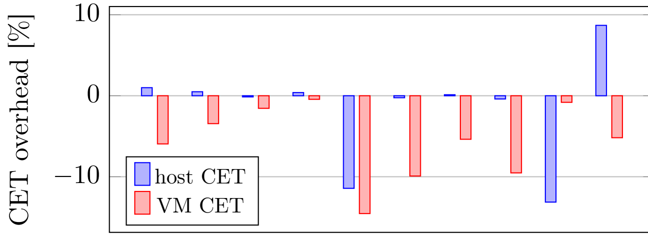
(c) large workload

Fig. 5.7: Processing time for the three workload sizes and test programs under four test scenarios

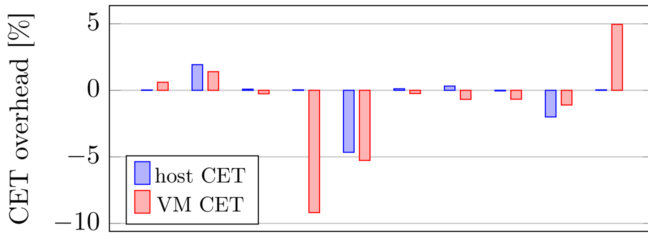
from a processing time when `awk` performs file-based operations. We do not infer from such comparisons.

We draw general conclusions from a Fig. 5.7, and then move on to a more detailed analysis. The most important observation here is that CET does not add a deterministic overhead to the processing time. In contrast, for most tests, we observed that the processing time decreased slightly when CET was used. This happened in all tests for small workloads, in 70% for medium workloads and in 60% tests for large workloads. This is clearly visible for applications such as `test-subroutine`, `filehash`, `convert`, or `bc` in the three workloads, where **VM: CET** bars are lower than the same test without CET in place. This behavior is observed regardless of whether a test was executed in bare metal or inside a virtual machine. However, in the case of a bare metal, the difference is less significant. Fig. 5.7 also shows a higher processing time for virtual machine scenarios as a result of virtualization overhead. This expected behavior is discussed in detail later in this section.

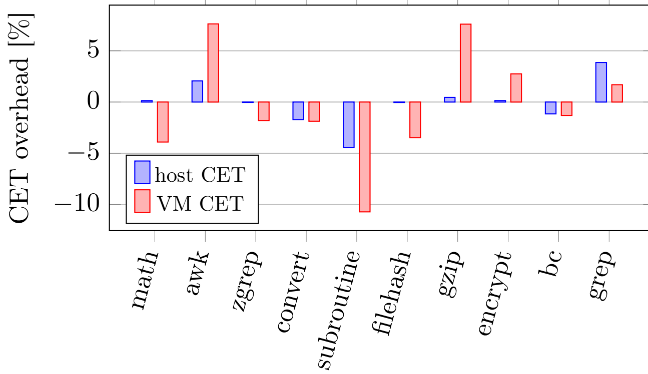
Initially, it may be assumed that performance will deteriorate due to the additional security introduced by CET. However, we prove that for CET this statement is not correct. This phenomenon can be clearly seen in Fig. 5.8 which presents the measured CET overhead separately for the three workloads. This overhead is calculated as a percentage difference in processing time between **Host: no CET** and **Host: CET** for a program running on a host and between **VM: no CET** and **VM: CET** for a program running inside a virtual machine. In Fig. 5.8, the bars are grouped into two for each test program. The blue bar shows a CET overhead for a program running on a host, whereas the red bar is the overhead captured inside a VM. The positive values on the y-axis in Fig. 5.8 show that a CET-enabled program was slower than the same program executed without CET protection. Negative values represent performance improvement for host and virtual machine scenarios. For some programs, we notice CET overhead inconsistency where an overhead is positive for a bare metal and negative for a virtual machine, or the opposite. This is visible for half of test programs (with the biggest anomaly `grep`) for small workloads and does not exist for large workloads. We notice that for small workloads, all test programs performed better inside a VM with CET protection in place. Test programs like `zgrep` or subroutine show consistent behavior regardless of the load. For the rest of the programs, the measured overhead varies for various workloads. As an example, `convert` is much faster for medium workloads, but for small workloads it does show any improvement. Some programs like `awk` show performance deterioration inside a VM when workload increases, but it is not observed for a bare metal. For the test program that we selected, despite the large scatter of values, we observe that the average CET improvement is always positive. This is presented in Fig. 5.9a. An improvement measured for a VM (red bars) varies between 0.4% and 14.5% with



(a) small workload

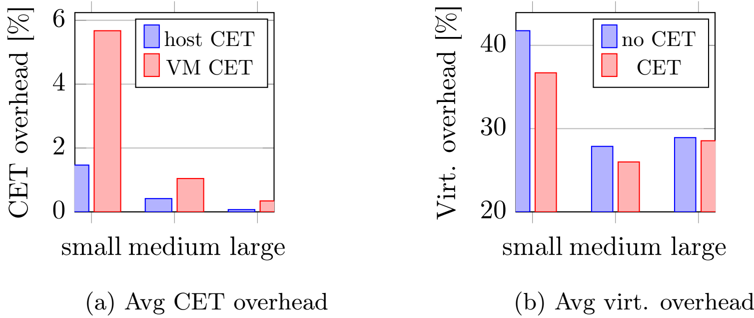


(b) medium workload



(c) large workload

Fig. 5.8: CET overhead for a bare metal and a virtual machine



(a) Avg CET overhead

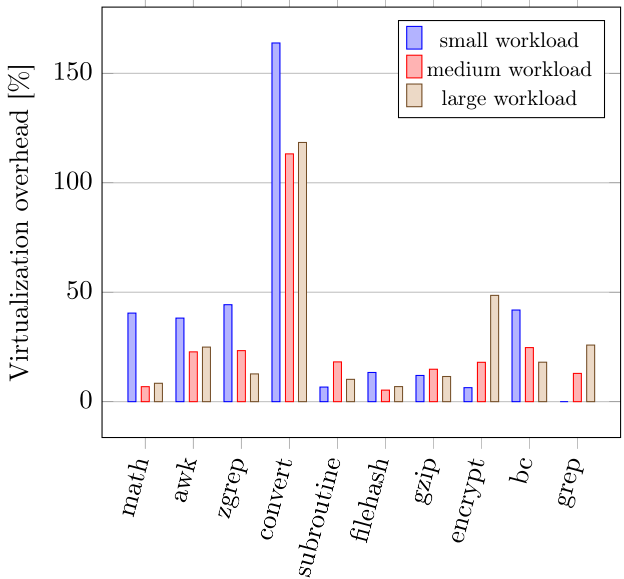
(b) Avg virt. overhead

Fig. 5.9: Avgerage CET and virtualization overhead

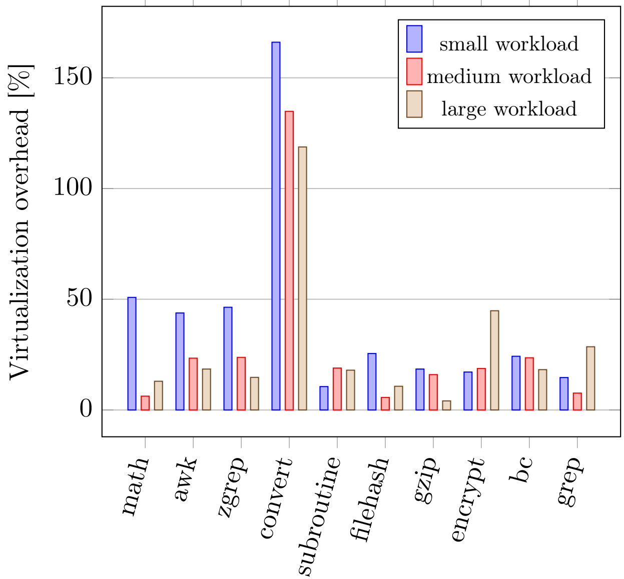
an average of 5.7% for a small workload. For medium workloads, an improvement varies between -5% and 9.2% with an average of 1.0%, while for large workloads the improvement lies between -7.6% and 10.7% with an average of 0.3%. Similar positive improvement is observed for a bare metal (blue bars), where an average is 1.5%, 0.4% and 0.1% for small, medium and large workloads respectively. It is observed in Fig. 5.9a that for larger workloads, performance improvement deteriorates. With this trend, a performance impact for a very large workloads can be negligible for both VM and bare metal.

In addition to the CET overhead, a major contributor to overall processing time is virtualization overhead. We show the virtualization overhead in two subgraphs in Fig. 5.10. In this case, a percentage overhead presented in Fig. 5.10a is calculated from the difference in processing time between **Host: CET** and **VM: CET** for a program protected with CET. For a program running without CET protection, in Fig. 5.10b, a percentage overhead is calculated from a difference between **Host: no CET** and **VM: no CET**. Both graphs show significant virtualization overhead. For all types of workload, the `convert` reports the highest virtualization overhead, reaching 166% for small workload. For other test programs, the overhead do not exceed 50%. We notice that in most of the cases (with the exception of `subroutine`, `encrypt` and `grep`), the overhead is highest for small workloads, which correspond to the blue bars in Fig. 5.10. For `grep` and `encrypt`, the highest values are reported for large workloads. One can also observe that `zgrep` and `grep` report different behaviors. For `zgrep` (which is used to search compressed files), the virtualization overhead decreases with load, while it increases for `grep`.

For small workloads, virtualization overhead varies between 10.6% and 166% with an average of 41.7%. For medium workloads, an overhead varies between 5.6% and 135% with an average of 27.9% and for large between 4.1% and 118.8% with an average of 28.9%. For CET protected programs, an average virtualization



(a) CET



(b) no CET

Fig. 5.10: Virtualization overhead

overhead for small, medium, and large workloads is 36.7%, 26% and 28.5%, respectively. The highest virtualization overhead is reported for small workloads. This can be explained by the fact that for not warmed-up system, a startup overhead (visible for small workloads) exists. For medium and large workloads, it takes much longer to process data than to start and warm-up, therefore a startup overhead is not visible. A virtualization overhead reported for medium workloads is lowest, and slightly higher for large workloads. We expect to see similar virtualization overhead for any larger workload executed. It is observed in Fig. 5.9b that the virtualization overhead is smaller for CET enabled scenarios (red bars). It is 2.5% lower on average in all test programs. This means that in a virtualized environment, a CET-enabled program performs better than the same software without hardware protection in place. From the performance perspective, running CET-disabled software inside a CET-enabled virtualized environment is suboptimal and should be avoided.

We proposed a solution for runtime attestation that can effectively enforce runtime integrity and prevent control-flow hijacking. The solution is fully integrated with our remote attestation for static resources, ensuring complete application integrity. The tests proved that for CET, the statement that performance will deteriorate due to the additional security introduced by CET is not correct. The phenomenon can be seen in the measured overhead separately for the three workloads. This makes a solution good candidate to consider for production environment.

6 Conclusions

The thesis statement presented in this dissertation in Sec. 1.1 is: *It is possible to assure static and runtime integrity for virtual machines that leverage Enhanced Platform Awareness capabilities in an untrusted cloud infrastructure with minimal changes to the existing software stack.* We have demonstrated that our **SGX+CET** solution can ensure integrity for virtual machines operating in untrusted infrastructure. We have shown that only minimal modifications to the existing software stack were necessary. With that changes, virtual machines can utilize the processor's Enhanced Platform Awareness capabilities for conducting static and runtime integrity checks. Integration of **SGX+CET** with the cloud components provides an additional layer of security without significant modifications to the application or infrastructure. As a result, our solution enhancing security in cloud-based environments can gain traction in the industry.

In this dissertation, we focus on integrity, which is one of the most important aspects of security. Ensuring system integrity is essential for maintaining the trust and confidence of users and customers. We present a novel solution for remote attestation, a process that can validate and enforce virtual machine integrity. With our **SGX+CET** solution, we address existing problems in this field that are crucial to the industry. We propose a comprehensive solution that can simultaneously provide the attestation for static system resources and runtime integrity by control-flow enforcement. We believe that one coherent proposal, fully owned and controlled by the application vendor, is more likely to be implemented than two independent solutions. Moreover, we eliminated complexity by reducing a TCB to CPU capabilities that are directly available for a virtual machine. We use a CPU as a hardware anchor to establish a chain-of-trust to the application, eliminating a problem with hardware modules virtualization. Another important factor is that the existing cloud stack does not require significant changes. Only a hypervisor update is necessary to support proper CPU features. We recognize

that this process is already happening for the SGX instruction set. We addressed aforementioned problems proposing a solution offering strong, hardware based security, that can fit into existing cloud computing ecosystem without negative impact on the complexity.

Evaluation of our proposal is another important part of this dissertation. It is clear that any production-ready solution has to perform well, scale and cannot negatively impact operations. We tackled this issue comprehensively. Initially, we assessed the processing overhead generated by our static attestation solution for files of different sizes. During analyzes, we found that files exceeding 10 kB showed a linear dependency between their size and processing time. Consequently, we derived a formula that could predict the measurement time for any given file size. Besides, we evaluated the performance and scaling capabilities of the attestation server, which are crucial for distributed virtual machines deployments on a large scale. We show that horizontal scaling does not introduce additional overhead and based on the number of virtual machines one can easily calculate the number of required attestation server instances. We also discovered areas of potential improvement that could enhance the processing capabilities of our solution. From a security perspective, we conducted an analysis of the threat model and examined SGX attack techniques, proposing countermeasures to mitigate potential vulnerabilities.

As a next step, we have introduced runtime attestation capabilities meeting expectations for a system that offers comprehensive integrity protection. Our proposal, does not depend on third-party components and provides a high level of hardware-based security with a negligible performance overhead.

The performance of the proposed runtime attestation extension, based on CET, was thoroughly verified. Our tests for a bare metal, based on the first available implementation, show promising results for hardware support for CFI. The measured 16% performance overhead is considered as a worst-case scenario that would only occur under exceptional circumstances and not in typical real-world applications. We demonstrate that the processing time overhead incurred by this technology for real applications is negligible. As a next step, we use CET technology, that was verified on a bare metal, as a foundation for a runtime attestation solution for virtual machines. We found that the processing overhead can be reduced when an operating system is compiled with CET support as the default protection mechanism against control-flow related attacks. For this reason, in any operating system that has CET capabilities, application vendors should also consider building their applications with hardware control flow enforcement.

Our performance tests also revealed that relying on CPU benchmarks that only measure pure arithmetic operations may yield incorrect results. Such test can lead to invalid conclusions, which is something we caution against.

Independently verified components of our system create a complementary

SGX+CET solution. It does not require any changes to the existing software, which makes it a good fit for the application vendors who can transit their services seamlessly. It eliminates some obstacle that exists in other implementation; therefore, it is a good candidate for cloud deployments.

In our future work, we aim to address adoption of our solution to containerized workloads. Containers have become increasingly popular due to their numerous advantages, including enhanced flexibility, efficiency, and scalability. The lightweight nature of containers offers several benefits, but also introduce more complexity that we should carefully consider in our model. While it remains unclear whether containers will entirely replace virtual machines in the future, it is visible in the industry that more virtual machines are deployed inside containers. This is primarily driven by the advantages we mentioned earlier. As a result, it is crucial for our solution to be compatible with both virtual machines and containers. With the additional study, we will understand any potential challenges or modifications needed to make our solution work seamlessly across both deployment models.

Bibliography

[1] DoD 5200.28-STD. *Trusted Computer System Evaluation Criteria.* Computer Security Center, December 1985.

[2] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[3] Tigist Abera, Nadarajah Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. C-FLAT: Control-flow Attestation for Embedded Systems Software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 743–754, 2016.

[4] Masoom Alam, Xinwen Zhang, Mohammad Nauman, Tamleek Ali, and Jean-Pierre Seifert. Model-based Behavioral Attestation. In *Proceedings of the 13th ACM symposium on Access control models and technologies*, pages 175–184, 2008.

[5] Masoom Alam, Tamleek Ali, Sanaullah Khan, Shahbaz Khan, Muhammad Ali, Mohammad Nauman, Amir Hayat, Muhammad Khurram Khan, and Khaled Alghathbar. Analysis of Existing Remote Attestation Techniques. *Security and Communication Networks*, 5(9):1062–1082, 2012.

[6] Moreno Ambrosin, Mauro Conti, Ahmad Ibrahim, Gregory Neven, Ahmad-Reza Sadeghi, and Matthias Schunter. SANA: Secure and Scalable Aggregate Network Attestation. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 731–742, 2016.

[7] Moreno Ambrosin, Mauro Conti, Riccardo Lazzeretti, Md Masoom Rabbani, and Silvio Ranise. Collective Remote Attestation at the Internet of Things

Scale: State-of-the-art and Future Challenges. *IEEE Communications Surveys & Tutorials*, 22(4):2447–2461, 2020.

[8] Mahmoud Ammar, Mahdi Washha, and Bruno Crispo. WISE: Lightweight Intelligent Swarm Attestation Scheme for IoT (the Verifier's Perspective). In *Proceedings of the 14th International Conference on Wireless and Mobile Computing, Networking and Communications (WiMob)*, pages 1–8. IEEE, 2018.

[9] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13. ACM New York, NY, USA, 2013.

[10] Sigurd Frej Joel Jørgensen Ankergård, Edlira Dushku, and Nicola Dragoni. State-of-the-art Software-based Remote Attestation: Opportunities and Open Issues for Internet of Things. *Sensors*, 21(5):1598, 2021.

[11] ARM. ARM Security Technology — Building a Secure System using TrustZone Technology. Technical report, ARM, 2009.

[12] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. A Security Framework for the Analysis and Design of Software Attestation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1–12, 2013.

[13] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation OSDI 16*, pages 689–703, 2016.

[14] Nadarajah Asokan, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter, Gene Tsudik, and Christian Wachsmann. Seda: Scalable Embedded Device Attestation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 964–975, 2015.

[15] NI Assurance. Hardware Control Flow Integrity CFI for an IT Ecosystem. *NSA: Fort Meade, MD, USA*, 2015.

[16] Alexander Sprogø Banks, Marek Kisiel, and Philip Korsholm. Remote Attestation: a Literature Review. *arXiv preprint arXiv:2105.02466*, 2021.

[17] Sébastien Bardin, Philippe Herrmann, and Franck Védrine. Refinement-based CFG Reconstruction from Unstructured Programs. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 54–69. Springer, 2011.

[18] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1213–1227, 2018.

[19] Henk Birkholz, Dave Thaler, Michael Richardson, Ned Smith, and Wei Pan. Remote ATtestation procedureS (RATS) Architecture. Technical report, RFC Editor. https://doi. org/10.17487/RFC9334. https://www. rfc-editor. org . . . , 2023.

[20] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33, 2017.

[21] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow Bending: On the Effectiveness of Control-flow Integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, 2015.

[22] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160, 2006.

[23] Binbin Chen, Xinshu Dong, Guangdong Bai, Sumeet Jauhar, and Yueqiang Cheng. Secure and Efficient Software-based Attestation for Industrial Control Devices with ARM Processors. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 425–436, 2017.

[24] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre Attacks: Stealing Intel secrets from SGX enclaves via speculative execution. *arXiv preprint arXiv:1802.09085*, 2018.

[25] Guoxing Chen, Yinqian Zhang, and Ten-Hwang Lai. OPERA: Open Remote Attestation for Intel's Secure Enclaves. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2317–2331, 2019.

[26] Liqun Chen, Rainer Landfermann, Hans Löhr, Markus Rohe, Ahmad-Reza Sadeghi, and Christian Stüble. A Protocol for Property-based Attestation.

In *Proceedings of the first ACM workshop on Scalable trusted computing*, pages 7–16, 2006.

[27] Liuhua Chen, Shilkumar Patel, Haiying Shen, and Zhongyi Zhou. Profiling and Understanding Virtualization Overhead in Cloud. In *2015 44th International Conference on Parallel Processing*, pages 31–40. IEEE, 2015.

[28] Quan Chen, Ahmed M Azab, Guruprasad Ganesh, and Peng Ning. Privwatcher: Non-bypassable Monitoring and Protection of Process Credentials from Memory Corruption Attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 167–178, 2017.

[29] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. StackArmor: Comprehensive Protection From Stack-based Memory Error Vulnerabilities for Binaries. In *Network and Distributed System Security*, 2015.

[30] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, Nadarajah Asokan, and Danfeng Yao. Exploitation Techniques for Data-oriented Attacks with Existing and Potential Defense Approaches. *ACM Transactions on Privacy and Security (TOPS)*, 24(4): 1–36, 2021.

[31] Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A Compile-time Solution to Buffer Overflow Attacks. In *Proceedings 21st International Conference on Distributed Computing Systems*, pages 409–417. IEEE, 2001.

[32] Sean Christopherson. Intel SGX Virtualization on Linux and KVM. In *KVM forum*, 2018.

[33] George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian O'Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. Principles of Remote Attestation. *International Journal of Information Security*, 10(2):63–81, 2011.

[34] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.

[35] Michael Coughlin, Eric Keller, and Eric Wustrow. Trusted Click: Overcoming Security Issues of NFV in the Cloud. In *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pages 31–36, 2017.

[36] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *USENIX security symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[37] Ang Cui. *Embedded System Security: A Software-based Approach.* PhD thesis, Columbia University, 2015.

[38] Thurston HY Dang, Petros Maniatis, and David Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 555–566, 2015.

[39] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-oriented Programming Attacks. In *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, pages 49–54, 2009.

[40] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-grained Control-flow Integrity Protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, 2014.

[41] Ivan de Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, and Gene Tsudik. Pure: Using Verified Remote Attestation to Obtain Proofs of Update, Reset and Erasure in Low-end Embedded Systems. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[42] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik. On the TOCTOU Problem in Remote Attestation. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2921–2936, 2021.

[43] Ghada Dessouky, Shaza Zeitouni, Thomas Nyman, Andrew Paverd, Lucas Davi, Patrick Koeberl, Nadarajah Asokan, and Ahmad-Reza Sadeghi. Lo-fat: Low-overhead Control Flow Attestation in Hardware. In *Proceedings of the 54th Annual Design Automation Conference 2017*, pages 1–6, 2017.

[44] Tu Dinh Ngoc, Bao Bui, Stella Bitchebe, Alain Tchana, Valerio Schiavoni, Pascal Felber, and Daniel Hagimont. Everything You Should Know About Intel SGX Performance on Virtualized Systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(1):1–21, 2019.

[45] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable Caches: Low-complexity Mitigation of Cache Side Channel Attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):1–21, 2012.

[46] Juan Du, Wei Wei, Xiaohui Gu, and Ting Yu. RunTest: Assuring Integrity of Dataflow Processing in Cloud Computing Infrastructures. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 293–304, 2010.

[47] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-grained Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 901–913, 2015.

[48] Aurélien Francillon, Quan Nguyen, Kasper B Rasmussen, and Gene Tsudik. A Minimalist Approach to Remote Attestation. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2014.

[49] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, 2018.

[50] Ali Gholami and Erwin Laure. Big Data Security and Privacy Issues in the Cloud. *International Journal of Network Security & Its Applications (IJNSA)*, 8(1):59–79, 2016.

[51] Dan Gonzales, Jeremy M. Kaplan, Evan Saltzman, Zev Winkelman, and Dulani Woods. Cloud-Trust—a Security Assessment Model for Infrastructure as a Service (IaaS) Clouds. *IEEE Transactions on Cloud Computing*, 5(3): 523–536, 2017. doi: 10.1109/TCC.2015.2415794.

[52] Trusted Computing Group. Virtualized Trusted Platform Architecture Specification, September 2011. URL `https://trustedcomputinggroup.org/wp-content/uploads/TCG_VPWG_Architecture_V1-0_R0-26_FINAL.pdf`. [Online; posted 27-September-2011].

[53] Grsecurity. How Does RAP Work. URL `https://grsecurity.net/rap_faq`.

[54] Jinyu Gu, Zhichao Hua, Yubin Xia, Haibo Chen, Binyu Zang, Haibing Guan, and Jinming Li. Secure Live Migration of SGX Enclaves on Untrusted Cloud. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 225–236. IEEE, 2017.

[55] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic Remote Attestation: a Virtual Machine Directed Approach to Trusted Computing. In *USENIX Virtual Machine Research and Technology Symposium*, volume 2004, 2004.

[56] Xiaofei He, Xinyu Yang, Rui Li, and Qingyu Yang. A Novel Delay-resilient Remote Memory Attestation for Smart Grid. In *International Conference on Wireless Algorithms, Systems, and Applications*, pages 88–99. Springer, 2013.

[57] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W Davidson, David Evans, John C Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. Secure and Practical Defense Against Code-injection Attacks Using Software Dynamic Translation. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 2–12, 2006.

[58] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. vTZ: Virtualizing ARM TrustZone. In *26th USENIX Security Symposium Security 17*, pages 541–556, 2017.

[59] Intel. Intel Software Guard Extensions SDK for Linux OS, July 2017. URL `https://download.01.org/intel-sgx/linux-1.9/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.9_Open_Source.pdf`. [Online; posted July-2017].

[60] Intel. KVM SGX, 2019. URL `https://github.com/intel/kvm-sgx`.

[61] William A Johnson, Sheikh Ghafoor, and Stacy Prowell. A Taxonomy and Review of Remote Attestation Schemes in Embedded Systems. *IEEE Access*, 9:142390–142410, 2021.

[62] Bernhard Kauer. OSLO: Improving the Security of Trusted Computing. In *USENIX Security Symposium*, volume 24, page 173, 2007.

[63] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A High-resolution Side-channel Attack on Last-level Cache. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[64] Florian Kelbert, Franz Gregor, Rafael Pires, Stefan Köpsell, Marcelo Pasin, Aurélien Havet, Valerio Schiavoni, Pascal Felber, Christof Fetzer, and Peter Pietzuch. Securecloud: Secure Big Data Processing in Untrusted Clouds. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 282–285. IEEE, 2017.

[65] Chongkyung Kil, Emre C Sezer, Ahmed M Azab, Peng Ning, and Xiaolan Zhang. Remote Attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence. In *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 115–124. IEEE, 2009.

[66] Steven L. Kinney. *Trusted Platform Module Basics: Using TPM in Embedded Systems*. Elsevier, 2006.

[67] Thomas Knauth, Michael Steiner, Somnath Chakrabarti, Li Lei, Cedric Xing, and Mona Vij. Integrating Remote Attestation with Transport Layer Security. *arXiv preprint arXiv:1801.05863*, 2018.

[68] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy SP'19*, pages 1–19. IEEE, 2019.

[69] Florian Kohnhäuser, Niklas Büscher, and Stefan Katzenbeisser. Salad: Secure and Lightweight Attestation of Highly Dynamic and Disruptive Networks. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 329–342, 2018.

[70] Kari Kostiainen, Nadarajah Asokan, and Jan-Erik Ekberg. Practical Property-based Attestation on Mobile Devices. In *International Conference on Trust and Trustworthy Computing*, pages 78–92. Springer, 2011.

[71] Nikos Koutroumpouchos, Christoforos Ntantogian, Sofia-Anna Menesidou, Kaitai Liang, Panagiotis Gouvas, Christos Xenakis, and Thanassis Giannetsos. Secure Edge Computing with Lightweight Control-flow Property-based Attestation. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 84–92. IEEE, 2019.

[72] Michał Kucab, Piotr Boryło, and Piotr Chołda. Remote Attestation and Integrity Measurements with Intel SGX for Virtual Machines. *Computers & Security*, 106:102300, 2021.

[73] Michał Kucab, Piotr Boryło, and Piotr Chołda. Performance Impact of Control Flow Enforcement Technology (CET). In *2022 25th Conference on Innovation in Clouds, Internet and Networks (ICIN)*, pages 96–100. IEEE, 2022.

[74] Michał Kucab, Piotr Boryło, and Piotr Chołda. Hardware-Assisted Static and Runtime Attestation for Cloud Deployments. *IEEE Transactions on Cloud Computing. Major revision request.*, 2023.

[75] D. Kuvaiskii. Add Exitless System Calls, 2018. URL https://github.com/oscarlab/graphene/pull/405.

[76] Markku Kylänpää and Aarne Rantala. Remote Attestation for Embedded Systems. In *Security of Industrial Control Systems and Cyber Physical Systems*, pages 79–92. Springer, 2015.

[77] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium Security 18*, pages 973–990, 2018.

[78] Dongxi Liu, Jack Lee, Julian Jang, Surya Nepal, and John Zic. A Cloud Architecture of Virtual Trusted Platform Modules. In *2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*, pages 804–811. IEEE, 2010.

[79] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating Last-level Cache Side Channel Attacks in Cloud Computing. In *2016 IEEE international symposium on high performance computer architecture (HPCA)*, pages 406–418. IEEE, 2016.

[80] Yutao Liu, Peitao Shi, Xinran Wang, Haibo Chen, Binyu Zang, and Haibing Guan. Transparent and Efficient CFI Enforcement with Intel Processor Trace. In *2017 IEEE International Symposium on High performance computer architecture (HPCA)*, pages 529–540. IEEE, 2017.

[81] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically Enforced Control Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 941–951, 2015.

[82] John Mechalas. Virtualizing Intel Software Guard Extensions with KVM and QEMU, May 2019. URL https://software.intel.com/content/www/us/en/develop/articles/virtualizing-intel-software-guard-extensions-with-kvm-and-qemu.html. [Online; posted 08-May-2019].

[83] Jämes Ménétrey, Christian Göttel, Anum Khurshid, Marcelo Pasin, Pascal Felber, Valerio Schiavoni, and Shahid Raza. Attestation Mechanisms for Trusted Execution Environments Demystified. In *Distributed Applications and Interoperable Systems: 22nd IFIP WG 6.1 International Conference, DAIS 2022, Held as Part of the 17th International Federated Conference on Distributed Computing Techniques, DisCoTec 2022, Lucca, Italy, June 13-17, 2022, Proceedings*, pages 95–113. Springer, 2022.

[84] Jämes Ménétrey, Christian Göttel, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. An Exploratory Study of Attestation Mechanisms for Trusted Execution Environments. *arXiv preprint arXiv:2204.06790*, 2022.

[85] Xiaozhu Meng and Barton P Miller. Binary Code is not Easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 24–35, 2016.

[86] Tanmaya Mishra, Thidapat Chantem, and Ryan Gerdes. Survey of Control-flow Integrity Techniques for Real-time Embedded Systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 21(4):1–32, 2022.

[87] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W Hamlen, and Michael Franz. Opaque Control-Flow Integrity. In *Network and Distributed System Security*, 2015.

[88] Amin Mosayyebzadeh, Apoorve Mohan, Sahil Tikale, Mania Abdi, Nabil Schear, Trammell Hudson, Charles Munson, Larry Rudolph, Gene Cooperman, Peter Desnoyers, and Orran Krieger. Supporting Security Sensitive Tenants in a Bare-Metal Cloud. In *Proceedings of the USENIX Annual Technical Conference*, pages 587–602, 2019.

[89] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. VRASED: A Verified Hardware/Software Co-Design for Remote Attestation. In *USENIX Security Symposium*, pages 1429–1446, 2019.

[90] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. Dialed: Data Integrity Attestation for Low-end Embedded devices. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 313–318. IEEE, 2021.

[91] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX Enclaves from Practical Side-channel Attacks. In *Proceedings of the 2018 USENIX Annual Technical Conference ATC 18*, pages 227–240, 2018.

[92] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: ExitLess OS services for SGX enclaves. In *Proceedings of the 12th European Conference on Computer Systems*, pages 238–253, 2017.

[93] Wojciech Ozga, Christof Fetzer, et al. TRIGLAV: Remote Attestation of the Virtual Machine's Runtime Integrity in Public Clouds. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 1–12. IEEE, 2021.

[94] Jaehong Park and Ravi Sandhu. Towards Usage Control Models: Beyond Traditional Access Control. In *Proceedings of the 7th ACM symposium on Access control models and technologies*, pages 57–64, 2002.

[95] Jaemin Park, Sungjin Park, Jisoo Oh, and Jong-Jin Won. Toward Live Migration of SGX-enabled Virtual Machines. In *2016 IEEE World Congress on Services (SERVICES)*, pages 111–112. IEEE, 2016.

[96] Jaemin Park, Sungjin Park, Brent Byunghoon Kang, and Kwangjo Kim. eMotion: An SGX Extension for Migrating Enclaves. *Computers & Security*, 80:173–185, 2019.

[97] Siani Pearson. Trusted Computing Platforms, the Next Security Solution. Technical report, HP Laboratories, 2002.

[98] Siani Pearson and Boris Balacheff. *Trusted Computing Platforms: TCPA Technology in Context*. Prentice Hall Professional, 2003.

[99] Ronald Perez, Reiner Sailer, Leendert van Doorn, Stefan Berger, Ramon Caceres, and Kenneth Goldman. VTPM: Virtualizing the Trusted Platform Module. In *Proc. 15th Conf. on USENIX Security Symposium*, pages 305–320, 2006.

[100] Sandro Pinto and Nuno Santos. Demystifying ARM Trustzone: A Comprehensive Survey. *ACM computing surveys (CSUR)*, 51(6):1–36, 2019.

[101] Sanndro Pinto, Hugo Araujo, Daniel Oliveira, Jose Martins, and Adriano Tavares. Virtualization on TrustZone-enabled Microcontrollers? Voilà! In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 293–304. IEEE, 2019.

[102] Marco Prandini and Marco Ramilli. Return-oriented Programming. *IEEE Security & Privacy*, 10(6):84–87, 2012.

[103] Weizhong Qiang, Yingda Huang, Hai Jin, Laurence T Yang, and Deqing Zou. CloudCFI: Context-Sensitive and Incremental CFI in the Cloud Environment. *IEEE Transactions on Cloud Computing*, 9(3):938–957, 2019.

[104] Sandeep Romana, Himanshu Pareek, and Lakshmi R. Dynamic Root of Trust and Challenges. *International Journal of Security, Privacy and Trust Management*, 5:01–06, 05 2016.

[105] Ahmad-Reza Sadeghi and Christian Stüble. Property-based Attestation for Computing Platforms: Caring about Properties, not Mechanisms. In *Proceedings of the 2004 workshop on New security paradigms*, pages 67–77, 2004.

[106] Ahmad-Reza Sadeghi, Christian Stüble, and Marcel Winandy. Property-based TPM Virtualization. In *International Conference on Information Security*, pages 1–16. Springer, 2008.

[107] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert Van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *USENIX Security symposium*, volume 13, pages 223–238, 2004.

[108] Paul Sangster, Lee Wilson, and Dean Liberty. Virtualized Trusted Platform Architecture Specification. *Specification Version*, 1, 2011.

[109] Nitin V. Sarangdhar and Daniel Nemiroff. Enhanced Privacy ID Based Platform Attestation, July 15 2014. US Patent 8,782,401.

[110] Muhammad Usama Sardar. Understanding Trust Assumptions for Attestation in Confidential Computing. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, pages 49–50. IEEE, 2022.

[111] Sarwar Sayeed and Hector Marco-Gisbert. On the Effectiveness of Control-flow Integrity against Modern Attack Techniques. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 331–344. Springer, 2019.

[112] Sarwar Sayeed, Hector Marco-Gisbert, Ismael Ripoll, and Miriam Birch. Control-flow Integrity: Attacks and Protections. *Applied Sciences*, 9(20), 2019. URL `https://www.mdpi.com/2076-3417/9/20/4229`.

[113] Vinnie Scarlata, Simon Johnson, James Beaney, and Piotr Zmijewski. Supporting Third Party Attestation for Intel SGX with Intel Data Center Attestation Primitives. *White Paper*, 2018.

[114] Cole Schlesinger, Karthik Pattabiraman, Nikhil Swamy, David Walker, and Benjamin Zorn. Modular Protections against Non-control Data Attacks. *Journal of Computer Security*, 22(5):699–742, 2014.

[115] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Abusing Intel SGX to Conceal Cache Attacks. *Cybersecurity*, 3(1):1–20, 2020.

[116] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. SWATT: Software-based Attestation for Embedded Devices. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pages 272–282. IEEE, 2004.

[117] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[118] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-space Randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, 2004.

[119] Ashish Singh and Kakali Chatterjee. Cloud Security Issues and Challenges: A Survey. *Journal of Network and Computer Applications*, 79:88–115, 2017.

[120] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: Hardware-assisted Data-flow Isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17. IEEE, 2016.

[121] Yuan Song, Wenchang Shi, Bo Qin, and Bin Liang. A Collective Attestation Scheme Towards Cloud System. *Cluster Computing*, pages 1–12, 2020.

[122] Rodrigo Vieira Steiner and Emil Lupu. Attestation in Wireless Sensor Networks: A Survey. *ACM Computing Surveys (CSUR)*, 49(3):1–31, 2016.

[123] Rodrigo Vieira Steiner and Emil Lupu. Towards more Practical Software-based Attestation. *Computer Networks*, 149:43–55, 2019.

[124] Haonan Sun, Rongyu He, Yong Zhang, Ruiyun Wang, Wai Hung Ip, and Kai Leung Yung. ETPM: A Trusted Cloud Platform Enclave TPM Scheme Based on Intel SGX Technology. *Sensors*, 18(11):3807, 2018.

[125] Yogesh Swami. In *Proc. Black Hat USA*, 2017.

[126] Henrik Theiling. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications*, pages 23–30. IEEE, 2000.

[127] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-edge Control-flow Integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, 2014.

[128] Flavio Toffalini, Eleonora Losiouk, Andrea Biondo, Jianying Zhou, and Mauro Conti. ScaRR: Scalable Runtime Remote Attestation for Complex Systems. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 121–134, 2019.

[129] Trusted Computing Group and others. TPM main specification level 2 version 1.2, revision 116, 2011.

[130] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution Through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy SP'20*, pages 1399–1417, 2020.

[131] Perry Wagle and Crispin Cowan. Stackguard: Simple Stack Smash Protection for GCC. In *Proceedings of the GCC Developers Summit*, pages 243–255, 2003.

[132] J. Wang, Z. Hong, Y. Zhang, and Y. Jin. Enabling Security-Enhanced Attestation With Intel SGX for Remote Terminal and IoT. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):88–96, Jan 2018. ISSN 1937-4151. doi: 10.1109/TCAD.2017.2750067.

[133] Juan Wang, Shirong Hao, Yi Li, Chengyang Fan, Jie Wang, Lin Han, Zhi Hong, and Hongxin Hu. Challenges Towards Protecting VNF with SGX. In *Proceedings of the 2018 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*, pages 39–42, 2018.

[134] Juan Wang, Chengyang Fan, Jie Wang, Yueqiang Cheng, Yinqian Zhang, Wenhui Zhang, Peng Liu, and Hongxin Hu. SvTPM: A Secure and Efficient vTPM in the Cloud. *arXiv preprint arXiv:1905.08493*, 2019.

[135] Zhenghong Wang and Ruby B Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 494–505, 2007.

[136] Zhenghong Wang and Ruby B Lee. A Novel Cache Architecture with Enhanced Performance and Security. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 83–93. IEEE, 2008.

[137] Johannes Winter. Trusted Computing Building Blocks for Embedded Linux-based ARM TrustZone Platforms. In *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pages 21–30, 2008.

[138] Liang Xu, Fangqi Sun, and Zhendong Su. Constructing Precise Control Flow Graphs from Binaries. *University of California, Davis, Tech. Rep*, 2009.

[139] Xiaoyang Xu, Masoud Ghaffarinia, Wenhao Wang, Kevin W Hamlen, and Zhiqiang Lin. CONFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1805–1821, 2019.

[140] Qiang Yan, Jin Han, Yingjiu Li, Robert H Deng, and Tieyan Li. A Software-based Root-of-trust Primitive on Multicore Platforms. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 334–343, 2011.

[141] Yi Yang, Xinran Wang, Sencun Zhu, and Guohong Cao. Distributed Software-based Attestation for Node Compromise Detection in Sensor Networks. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 219–230. IEEE, 2007.

[142] Charles Yount, Harish Patil, Mohammad S Islam, and Aditya Srikanth. Graph-matching-based Simulation-region Selection for Multiple Binaries. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 52–61. IEEE, 2015.

[143] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. Atrium: Runtime Attestation Resilient Under Memory Attacks. In *Proceedings of the 2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 384–391. IEEE, 2017.

[144] Jiliang Zhang, Binhang Qi, Zheng Qin, and Gang Qu. HCIC: Hardware-assisted Control-flow Integrity Checking. *IEEE Internet of Things Journal*, 6(1):458–471, 2018.

Index

application, vi, xv, 1–3, 5, 7–12, 14–21, 23, 24, 26–32, 37, 38, 40, 42–46, 48, 50, 53–55, 62, 64, 65, 67–69, 71, 74, 77, 78, 82, 86–89

bare metal, v, xv, 1, 2, 5, 7, 8, 11, 18, 23, 25, 28, 38, 53, 54, 60, 62, 67, 70, 71, 78, 82–84

chain-of-trust, xv, 3, 10, 19, 20, 24, 34, 75, 87
cloud computing, xv, 1, 2, 88
cloud software stack, xv, 26, 64, 77

enclave, xv, 14–16, 24, 25, 31, 32, 38–40, 42–44, 46–50, 53, 54, 56, 60, 62–64
EPA, xiii, xvi, 3, 87

host, xvi, 3, 19, 20, 26, 31, 38, 45, 54, 73–75, 77, 78, 82
HSM, xiii, xvi, 44
hybrid cloud, xvi, 8
hyperscaler, xvi

IaaS, xiii, xv, xvi, 3, 8, 19, 26, 28, 43, 74
integrity, vi, xvi, 2, 3, 5, 6, 8–12, 15, 16, 18–21, 23–26, 28, 30, 34, 35, 43–46, 53, 55, 62, 64, 65, 67, 68, 73–75, 77, 86–88

MaaS, xiii, xvi, 8, 43
multi-cloud, xvii, 7, 8, 21, 38

PaaS, xiii, xv, xvii
PCR, xiii, xvii, 20, 23, 24, 28, 45, 46, 75
private cloud, xvii
public cloud, xvii, 8, 9, 44, 46

root-of-trust, xvii, 3, 10, 23–26, 34, 35, 43, 72

SaaS, xvii
server, xvii, 8, 11, 25, 36, 39, 40, 42–44, 47, 50, 51, 63, 88
service, xvii, 7, 21, 31, 38–40, 42, 45, 62
service quality, xvii, 7

TCB, xiv, xviii, 10, 21, 39, 45, 46, 87
TCG, xiv, xviii, 7, 19, 23
TEE, xiv, xviii, 18, 67
TPM, xiv, xviii, 7, 19, 20, 23, 24, 26–28, 35, 44–46, 67, 74, 75

trusted computing, xviii, 3, 8, 18, 20, 34, 37, 45

virtualization, xviii, 1–3, 5, 7, 10, 25, 53, 56, 60, 78, 80, 82, 84, 86, 87

VNF, xiv, xviii, 46