TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Systems Architecture, Chair of Systems Engineering

Hardening High-Assurance Security Systems with Trusted Computing

Wojciech Ozga

Dissertation

to achieve the academic degree

Doktoringenieur (Dr.-Ing.)

Advisor
Dr.-Ing. Silvio Dragone

Supervisor
Prof. Dr. Christof Fetzer

Submitted on: 01.11.2021
Defended on: 29.06.2022

For my parents

Abstract

We are living in the time of the digital revolution in which the world we know changes beyond recognition every decade. The positive aspect is that these changes also drive the progress in quality and availability of digital assets crucial for our societies. To name a few examples, these are broadly available communication channels allowing quick exchange of knowledge over long distances, systems controlling automatic share and distribution of renewable energy in international power grid networks, easily accessible applications for early disease detection enabling self-examination without burdening the health service, or governmental systems assisting citizens to settle official matters without leaving their homes. Unfortunately, however, digitalization also opens opportunities for malicious actors to threaten our societies if they gain control over these assets after successfully exploiting vulnerabilities in the complex computing systems building them. Protecting these systems, which are called *high-assurance security systems*, is therefore of utmost importance.

For decades, humanity has struggled to find methods to protect high-assurance security systems. The advancements in the computing systems security domain led to the popularization of hardware-assisted security techniques, nowadays available in commodity computers, that opened perspectives for building more sophisticated defense mechanisms at lower costs. However, none of these techniques is a silver bullet. Each one targets particular use cases, suffers from limitations, and is vulnerable to specific attacks. I argue that some of these techniques are synergistic and help overcome limitations and mitigate specific attacks when used together. My reasoning is supported by regulations that legally bind high-assurance security systems' owners to provide strong security guarantees. These requirements can be fulfilled with the help of diverse technologies that have been standardized in the last years.

In this thesis, I introduce new techniques for hardening high-assurance security systems that execute in remote execution environments, such as public and hybrid clouds. I implemented these techniques as part of a framework that provides technical assurance that high-assurance security systems execute in a specific data center, on top of a trustworthy operating system, in a virtual machine controlled by a trustworthy hypervisor or in strong isolation from other software. I demonstrated the practicality of my approach by leveraging the framework to harden real-world applications, such as machine learning applications in the eHealth domain. The evaluation shows that the framework is practical. It induces low performance overhead (<6%), supports software updates, requires no changes to the legacy application's source code, and can be tailored to individual trust boundaries with the help of security policies.

The framework consists of a decentralized monitoring system that offers better scalability than traditional centralized monitoring systems. Each monitored machine runs a piece of code that verifies that the machine's integrity and geolocation conform to the given security policy. This piece of code, which serves as a *trusted anchor* on that machine, executes inside the trusted execution environment, *i.e.*, Intel software guard extensions (SGX) [45], to protect itself from the untrusted host, and uses trusted computing techniques, such as trusted platform module (TPM) [90], secure boot, and integrity measurement architecture (IMA) [225, 89], to attest to the load-time and runtime integrity of the surrounding operating system running on a bare metal machine or inside a virtual machine. The trusted anchor implements my novel, formally proven protocol, enabling detection of the TPM cuckoo attack [205].

The framework also implements a key distribution protocol that, depending on the individual security requirements, shares cryptographic keys only with high-assurance security systems executing in the predefined security settings, *i.e.*, inside the trusted execution environments or inside the integrity-enforced operating system. Such an approach is particularly appealing in the context of machine learning systems where some algorithms, like the *machine learning model training*, require temporal access to large computing power. These algorithms can execute inside a dedicated, trusted data center at higher performance because they are not limited by security features required in the shared execution environment. The evaluation of the framework showed that training of a machine learning model using real-world datasets achieved 0.96× native performance execution on the GPU and a speedup of up to 1560× compared to the state-of-the-art SGX-based system.

Finally, I tackled the problem of software updates, which makes the operating system's integrity monitoring unreliable due to false positives, *i.e.*, software updates move the updated system to an unknown (untrusted) state that is reported as an integrity violation. I solved this problem by introducing a proxy to a software repository that *sanitizes* software packages so that they can be safely installed. The sanitization consists of predicting and certifying the future (after the specific updates are installed) operating system's state. The evaluation of this approach showed that it supports 99.76% of the packages available in Alpine Linux® main and community repositories.

The framework proposed in this thesis is a step forward in verifying and enforcing that high-assurance security systems execute in an environment compliant with regulations. I anticipate that the framework might be further integrated with industry-standard security information and event management (SIEM) tools as well as other security monitoring mechanisms to provide a comprehensive solution hardening high-assurance security systems.

Acknowledgments

First and foremost, I would like to express the most immense gratitude to my supervisor Prof. Christof Fetzer for giving me space to explore new ideas and concepts and for openness to my individual decisions. I am sincerely thankful to Dr. Silvio Dragone for lighting the hidden corridors of the IBM Zurich lab; without him, I would have undoubtedly missed the ones I have taken.

I am grateful to the IBM Research GmbH, Zurich Research Laboratory, for building and providing me with a thought-provoking working place. I thank my advisor Dr. Silvio Dragone and my managers, Michael Osborne and Dr. Marc Ph. Stoecklin, for opening a new door in my career despite the ongoing pandemic and related difficulties. Thanks to all the group and lab members, particularly Dr. Patricia Sagmeister and Dr. Tamas Visegrady. I also thank Anne-Marie Cromack from the publication department of the IBM Research GmbH for reviewing this thesis.

At the same time, I would like to thank the members of the Systems Engineering Group at TU Dresden for their cooperation. Especially I am indebted to Do Le Quoc and Rasha Faqeh for their hard work put into our papers y a Gabriel Fernandez por fazer a vida em Dresden mais colorida e agradável, obrigado! Special thanks to Robert Krahn for support and Oleksii Oleksenko and Bohdan Trach for scientific discussions and advice. Last but not least, many thanks to Irina Karadschow, Claudia Einer, and Andrea Eulitz for building a level of indirection between me and the depths of the university.

I want to thank the reviewers of my thesis, especially Prof. Lianying (Viau) Zhao and Prof. Christof Fetzer, for their insightful comments and suggestions.

Muchas gracias a Araceli, Andrzej por vuestro apoyo en seguir luchando con la tesis. Ich bedanke mich bei Larisa, Martin und alle Freunde in Halle (Saale) für die tolle Zeit, die vom harten akademischen Alltag ablenkte. Przede wszystkim dziękuję moim rodzicom, siostrom, oraz Izabeli za wsparcie, cierpliwość oraz motywację. Dzięki Wam miałem przywilej kierować moje życie na różne ścieżki oraz robić krok wstecz by potem móc zacząć jeszcze raz lecz tym razem bogatszy o nowe doświadczenia.

Publications

The content of this thesis is based on the following publications.

(i) CHORS: Hardening High-assurance Security Systems With Trusted Computing. Wojciech Ozga, Rasha Faqeh, Do Le Quoc, Franz Gregor, Silvio Dragone, and Christof Fetzer. In the Proceedings of the 37th ACM Symposium On Applied Computing (SAC '22), 2022.

(ii) TRIGLAV: Remote Attestation of the Virtual Machine's Runtime Integrity in Public Clouds. Wojciech Ozga, Do Le Quoc, and Christof Fetzer. In Proceedings of the 2021 IEEE International Conference on Cloud Computing (CLOUD '21), 2021.

(iii) PERUN: Confidential Multi-Stakeholder Machine Learning Framework with Hardware Acceleration Support. Wojciech Ozga, Do Le Quoc, and Christof Fetzer. In Proceedings of the 35th Annual IFIP Working Conference on Data and Applications Security and Privacy (DBSec '21), 2021.

(iv) A Practical Approach for Updating an Integrity-Enforced Operating System. Wojciech Ozga, Do Le Quoc, and Christof Fetzer. In Proceedings of the 21st International ACM/IFIP Middleware Conference (Middleware '20), 2020.

Contents

List of Figures

List of Tables

Glossary

AI artificial intelligence. 67

AIK attestation key. 35, 43

API application programming interface. 36, 94, 95, 107

BIOS basic input/output system. 92

CA certificate authority. 29, 43, 53, 114

CDN content delivery network. 86

CICD continuous integration and continuous deployment. 94

CNN convolutional neural network. 78

CPU central processing unit. 97

DC data center. 25–27, 31, 33, 37, 44

DMA direct memory access. 17

DNN deep neural networks. 73

DRAM dynamic random-access memory. 14, 21

DRTM dynamic root of trust for measurements. 14–17, 24, 30, 44, 49, 52, 57, 113, 116

dTPM discrete TPM chip. 59, 60

ECDSA elliptic curve digital signature algorithm. 41, 60

EK endorsement key. 35

EPC enclave page cache. 21, 39, 59, 77, 79, 80, 97

EU European Union. 2

FIPS federal information processing standard publication. 12

GDPR general data protection regulation. 2, 5, 67

GPU graphical processing unit. 68, 70, 72, 74

HMAC hash-based message authentication code. 41

HSM hardware security module. 12

IAS Intel attestation service. 23, 58

IBM ACS IBM TPM attestation client-server. 43, 114

IMA integrity measurement architecture. 17, 24, 28, 44, 46, 47, 49, 52, 56, 63, 69, 70, 73, 76, 112, 114, 116, 117, II

Intel CIT Intel open cloud integrity technology. 43, 106, 113, 114

IOMMU input-output memory management unit. 49

IP intellectual property. 19

iTPM integrated TPM. 59, 60

KMS key management system. 37

KVM kernel-based virtual machine. 56, 115

LUKS Linux unified key setup. 19, 77

MC monotonic counter. 50, 53, 57, 58, 60, 96

MCS monotonic counter service. 57–60, 63

MitM man-in-the-middle. 50, 52–54

MKTME Intel multi-key total memory encryption. 64

ML machine learning. 40, 67, 70, 71, 81

MME memory management engine. 14

MPC multi-party computation. 68, 80

MRENCLAVE enclave hash measurement. 22, 23

NIC network interface card. 59, 97

NVRAM non-volatile random-access memory. 15

PAX portable archive exchange. 95

PCIe peripheral component interconnect express. 19

PCR platform configuration register. 15, 29, 35, 36, 43, 49, 109

PEF IBM protected execution facility. 20, 47

PTT Intel platform trusted technology. 59

QEMU quick emulator. 59

REST representational state transfer. 36, 94, 95, 107

ROM read-only memory. 15, 16, 19

RSA Rivest-Shamir-Adleman. 41

SCMMS security configuration management and monitoring service. 107–110

SEV AMD secure encrypted virtualization. 20, 47, 64, 65

SGX Intel software guard extensions. 20–23, 30, 31, 35, 37, 46, 47, 56–60, 64, 70, 74, 84, 92, 94, 96, 97, 112, 115, 116, II

SIEM security information and event management. 2, 32, 118, II

SLOC source lines of code. 64, 95

SMM system management mode. 21, 38

SoC system on chip. 19

SR-IOV single root input/output virtualization. 59

SSH secure shell. 47, 52, 54

SVM AMD secure virtual machine. 17

TCB trusted computing base. 6, 20, 21, 39, 56, 64, 72, 78, 80, 118

TCG Trusted Computing Group. 3, 14, 17, 43, 44, 114

TCTs trusted computing techniques. 12–14, 16, 20, 22, 69, 70, 76, 82, 115, 117

TDX Intel trust domain extensions. 20, 64, 65

TEE trusted execution environment. 11, 20, 24, 26, 37, 46, 47, 50–54, 56, 64, 65, 68, 84, 86, 92, 105, 115–118

TLS transport layer security. 43, 50, 53, 58, 60, 65, 74

TOCTOU time of check to time of use. 96

TOFU trust on first use. 108–110

TPM trusted platform module. 4, 13–15, 24, 29, 36, 43, 44, 46, 48–50, 52, 53, 56, 58–60, 63, 65, 70, 73, 89, 96, 109, 113–116, II

TPU tensor processing unit. 73

TXT Intel trusted execution technology. 17, 35, 57, 59, 75, 113

UEFI unified extensible firmware interface. 35

VM virtual machine. 46–54, 63, 65, 66

VPN virtual private network. 82

VT-d Intel virtualization technology for directed I/O. 59

vTPM virtual TPM. 46, 47, 49, 50

1 Introduction

1.1 Progressing Digitalization and Threats

These days computing systems support our everyday life on virtually every level. However, not all of us realize that our life, prosperity, and geopolitical stability heavily depend on some of these systems, further referred to as *high-assurance security systems* [1]. Some examples of these are systems processing our medical health records [138], banking applications managing our money [144], key management systems protecting our credentials [37, 159, 88], or governmental systems storing privacy-sensitive citizens' data. Data leaked from such systems might be used for blackmail, identity theft, or manipulation of democratic elections with the help of political preference profiling, just to name a few cases. Essentially, many of these systems form part of our critical infrastructure, like computing systems controlling hospitals, nuclear plants, and traffic light systems; telecommunication systems providing the backbone for security-critical information exchange; or water supply systems including secure water treatment plants and water distribution systems. Deviation from the expected, normal behavior of high-assurance security systems, resulting from sabotage or a successful hacker attack, might result in a disaster exposing humans' life to risk. Therefore, we must ensure that we protect these systems in order to guarantee peace and stability to our societies by harnessing all existing knowledge and technology.

Due to the security- and safety-critical character of high-assurance security systems, they present an attractive target for malicious actors, such as cybercriminals and governmentally motivated hackers. Cyber attacks against such systems have occurred in the past and will continue to occur, given the rapid digitalization of our societies. To name a few examples, in 2021, hackers successfully conducted a ransomware attack on a major US oil pipeline, which supplies 45% of the East Coast's fuel [49]. The incident resulted in the pipeline shutdown and a shortage of fuel in the eastern part of the United States. Although protections against ransomware attacks emerged [284, 29], they proved not be efficient. Security analytics predict that increasing popularity of ransomware attacks (20% of all incidents in 2019, 23% in 2020) will persist through 2021 [114]. In 2017, hackers managed to tamper with the emergency shutdown system in a Saudi petrochemical plant [250]. Although the attack resulted only in the plant shut down, it could have led to an accident [59]. In recent years,

[1] I refer to the high-assurance security system as hardware, software, and workload providing security-sensitive functionality to society.

malicious actors have attempted to infiltrate European and the United States nuclear power stations [251, 219]. A control gained over software controlling nuclear power stations' cooling systems might have allowed the hackers, for example, to cause power plant failure or even nuclear disaster similar to the one in Fukushima [16]. Successful cyber attacks resulting in leakage of users' data are not uncommon. Only in 2020, due to lack of proper protection, privacy-sensitive data of hundreds of millions of Brazilian citizens, including their sensitive health records, were leaked [30]. Yet another attack in 2020, this time against United States government agencies and companies, happened due to malicious changes to the source code of a network monitoring software distributed via a legitimate update procedure [247]. It allowed attackers to penetrate American's sensitive systems on an unprecedented scale. After many months, the scope of this attack is still unknown, but it is suspected that it could lead to leakage of confidential data.

1.2 Regulations as a Remedy?

Governments force legal entities owning high-assurance security systems to follow strict regulations [62, 249, 14, 61, 63, 77, 233] that define what security measures they must implement to isolate high-assurance security systems from potential threats; thus, preventing leakage of confidential data and ensuring correct system behavior.

For example, Germany defines protection mechanisms [77] that high-assurance security systems must implement to shield the privacy-sensitive data in the eHealth systems [76]. In particular, the high-assurance security system's owner must protect physical resources by enclosing machines inside video monitored security cages in an access-controlled data center. At the software level, the operating system must employ techniques to ensure software integrity. At the same time, the individual processes handling privacy-sensitive data must run in isolation from the operating system and the operator. Similarly, the European Union (EU) regulates the financial market and critical infrastructure [61, 62, 233]. The regulations require restricting physical and remote access to machines to limited personnel. Network, software, and access control must be constantly monitored, allowing timely response in case of anomaly detection. Moreover, a dedicated automated system, *e.g.*, SIEM, must correlate network and system alerts to detect multifaceted attacks.

At the European level, the general data protection regulation (GDPR) [63] also restricts the geographical location where privacy-sensitive data can be processed, *i.e.*, European citizens' personal and medical data must never leave the EU and cannot be disclosed to anyone without the citizen's approval [2]. Violating the regulations might result in a fine of up to 20M euros or 4% of the company's turnover in the preceding financial year. Like this, regulators force system providers and operators to implement respective countermeasures that eventually increase the resistance of high-assurance security systems to data leakage. However, ensuring that the software processing the data executes in the given geolocation, *i.e.*, on a computer in the specific data center, especially in the face of powerful adversaries that might trick the GPS signals, is not a straightforward task.

Notably, regulators do not define how specific requirements must be implemented, leaving system providers freedom in selecting and adjusting the existing technologies to their individual use cases. From the security perspective, a naïve combination of different security techniques does not necessarily provide more protection than using them individually. This

[2] More precisely, the data can be exchanged with countries outside EU but these countries must provide at least equivalent levels of data protection as the GDPR.

is because of the differences in their designs, threat models, and offered security guarantees. It usually requires expert knowledge to determine if and how these technologies could be combined to meet certain security guarantees imposed by regulations. As such, regulations are just a step in the right direction, but without reasonable design and implementation they provide little benefit.

1.3 Theory Meets Practice

The advancement of security techniques achieved within the last two decades has become the backbone for security solutions to meet the strict regulation requirements. The recent technologies known as *trusted execution environments* [169, 45, 184] are particularly important because they promise to protect individual applications against compromised operating systems controlled by rogue system administrators. It means that, at least theoretically, we might execute high-assurance security systems inside the trusted execution environment and stop worrying about the existing threats jeopardizing operating systems and the operators controlling them. This is not enough in practice, however. An application executing in the trusted execution environment by definition cannot exist without an operating system, which manages computing resources allocation and controls the application's life cycle. It means that an untrustworthy operating system might jeopardize the high-assurance security system executed in the trusted execution environment because it might shut down the system or run malware that would extract confidential data processed inside the application via microarchitectural or side-channel attacks [260, 277, 189]. Therefore, a *trustworthy operating system* is a key element of each high-assurance security system because it protects the application's safety and security. Thus, despite being very attractive in terms of performance and security guarantees, I argue that trusted execution environments should be accompanied by other security techniques that prevent or at least detect untrusted operating system states. My reasoning was supported in October 2019 by German policymakers who defined that German eHealth systems should rely on both concepts to protect German citizens' data [77].

A way to attest to the operating system's trustworthiness is by benefiting from widely adopted, standardized security techniques known as *trusted computing* [90, 235, 225] (not to be confused with *trusted execution* or *confidential computing*) developed by the not-for-profit Trusted Computing Group [256] organization. Trusted computing techniques ensure that only legitimate, certified software executes on a computer. However, because of the differences in designs, threat models, and security guarantees, it is an open question whether it is feasible and, if yes, how could the *trusted execution environment* integrate with *trusted computing* techniques? Would their combination lead to increased security and at what cost? Addressing these questions nowadays becomes more and more important because of the incoming regulations, such as above-mentioned German eHealth regulations [77].

1.4 Establishing Trust in a Remote Computer

Before we provision a remote computer with the confidential data, we must ensure that the high-assurance security system, which will process these data, is controlled by the expected operating system running on a computer located in the desired data center, according to the applicable legal regulations. This is not a trivial task as we cannot be sure that the com-

puter, with which we are communicating, is not controlled by an attacker who impersonates the legitimate computer. An attacker could take over the control of the computer by exploiting computer misconfiguration, using social engineering, or redirecting us to a machine under her control. By controlling the operating system, she would have enough capabilities to convince us that we are interacting with a legitimate computer. Thus, we need a technical assurance that the computer we communicate with is legitimate.

In chapter 3, I tackle the problem of establishing trust in a remote computer. I show that trusted computing techniques, which were designed to solve this problem, are not enough because they are vulnerable to the cuckoo attack [205]. I introduce a novel, practical, and formally proven defense mechanism against the cuckoo attack that relies on trusted computing and trusted execution environment techniques. I implement this defense mechanism as part of the framework that monitors and enforces the integrity of high-assurance security systems distributed among computers in data centers. I evaluate the framework while protecting a real-world eHealth application (subsection 3.8.1).

The framework establishes trust in a remote computer by first deploying a piece of trusted software (agent) inside the trusted execution environment on a potentially malicious remote computer. The agent ensures that the computer is in the expected data center and then establishes trust with a secure element, like TPM [90], attached to this computer. With the help of the secure element, the agent extends trust to the operating system using trusted computing. Eventually, we establish trust with the agent, which certifies that the computer is legitimate. Only then, we execute the high-assurance security system and provide it with secrets and confidential data. In chapter 7, I extend the framework with the configuration management and integrity monitoring system that leverages this technique at scale, allowing security officers to easily provision multiple computers, define expected integrity states, and continuously monitor the integrity state.

The proposed approach gives us an important primitive. It allows for trust to be established in an operating system running on a remotely accessible bare-metal computer. However, modern applications are frequently split into smaller services that execute inside disjoined virtual machines to utilize computing resources more effectively and simplify their management. The natural question that arises is that since we can now establish trust in the operating system running on a bare-metal computer, could we further extend trust in software executing inside virtual machines hosted on that computer?

1.5 Extending Trust to Virtual Machines

The cloud computing paradigm relies heavily on virtualization to dynamically allocate resources (in the form of virtual machines) on shared computing resources. It is beneficial for applications that require disjoined execution environments hosted on a single physical machine or large computing power for a limited amount of time. For example, consider the eHealth application that provides the electronic receipt functionality, as defined in the German eRezept specification [76]. Such a system requires more computing resources during the day when it is used by doctors, patients, and pharmacies, than during the night when it is barely used. From the economic point of view, it makes little sense to keep all computing resources up and running during low activity time. Instead, the cloud computing paradigm allows computing resources to be dynamically acquired or released depending on the application's needs and shares the resources with other systems or businesses that currently need them.

In chapter 4, I introduce a protocol to establish trust in a virtual machine running on a remote computer. In contrast to bare-metal computers, virtual machines require additional software managing computing resources. Such software, its configuration, and administration remain under the control of the system administrator, who must be trusted to behave legally. I show that by using the trusted computing techniques, I can effectively limit the system administrator's capabilities while leveraging the trusted execution environment to establish and maintain trust in the virtual machine runtime integrity of the software and its configuration. The proposed protocol is transparent to the virtual machine configuration and setup. It performs an implicit attestation of virtual machines during a secure login and binds the virtual machine integrity state with the secure connection. To demonstrate the practicality of the approach and gain insight into the performance, I built its prototype using state-of-the-art technologies commonly used in the cloud. The evaluation performed on real-world applications shows that the approach is practical and incurs reasonable performance overhead ($\leq$ 6%).

So far, we have focused on scenarios where the combination of *trusted computing* and *trusted execution environment* techniques is advantageous for systems requiring an increased level of security. However, certain application owners, like businesses running compute-intensive artificial intelligence algorithms, might prefer to trade-off some security guarantees to gain better performance. Thus, the practical approach should grant the flexibility to select the level of security. The open question is how to enable it for a general-purpose computation?

1.6 Adding Support for Hardware Accelerators

Over the last few years, big data and artificial intelligence have received a lot of attention due to advancements in the development of high-performance computing systems. It led to the creation of many valuable services enhancing our everyday life. For example, machine learning algorithms support doctors in recognizing brain tumors from magnetic resonance imaging scans [60], saving humans' lives by reducing the probability of false negatives. Considering that the European Union positions health and artificial intelligence as fundamental topics in its strategic plan for years 2021-2024 [64], we might expect similar systems to develop in the near future. For example, the EU4Health program focuses on improving cancer prevention, control, and care. At a large scale, these objectives might only be satisfied with the help of digital systems directly exposed to patients because only such systems might support a fast and inexpensive way to provide early detection of diseases. An example of such an early prevention system would allow citizens to upload their skin photos directly to an eHealth service that would use artificial intelligence algorithms to verify against melanoma cancer.

Artificial intelligence algorithms must be trained on real data to build such services. It typically requires access to large computing power but only for the duration of the computations. It is, therefore, reasonable to run such computations in the cloud, and only pay for the utilized resources. However, the training algorithms in the eHealth domain fall under regulations, such as GDPR [63], because they operate on privacy-sensitive data. State-of-the-art solutions, such as fully homomorphic encryption [78] or trusted execution environments [167], preserve data confidentiality but suffer from large performance overhead, which limits their practical application [263, 199]. Specifically, trusted execution environments involve significant performance degradation while processing a large amount of data — a typical machine

learning training model's scenario. This is because of a limited amount of secure memory available for the computation, lack of trusted input-output paths to hardware accelerators, and lack of support for respective trusted execution environments inside hardware accelerators. I notice, however, that with the help of trusted computing techniques, users might securely access hardware accelerators under additional security assumptions.

In chapter 5, I introduce a framework that enables users to trade-off between security and performance when executing machine learning computations. For example, a user can execute compute-intensive machine learning training workloads on hardware accelerators while relying on trusted computing to ensure the trustworthiness of the remote computer located in the trusted data center. Conversely, he would execute less compute-intensive workloads, such as inference, inside the trusted execution environment, and thus at a lower trusted computing base and stronger isolation. The evaluation shows that during the machine learning training on CIFAR-10 [157] and real-world medical datasets [236], the framework achieved a 161× to 1560× speedup compared to the pure trusted execution environment-based approach [167].

1.7 Enabling Updates of Integrity-Enforced Operating Systems

To enable the application of trusted computing techniques at scale, we must solve the problem of software updates. Specifically, integrity-enforced operating systems running in production cannot be updated because the integrity monitoring becomes unreliable due to the high number of false positives. Software update triggers an integrity violation alarm because the monitoring system detects unknown integrity measurements corresponding to updated software.

I address this problem by adding an extra level of indirection between the operating system and software repositories. In chapter 6, I introduce a software update repository proxy that overcomes the shortcomings of previous approaches by *sanitizing* software packages. The sanitization consists of modifying unsafe installation scripts and adding digital signatures in a way that software packages can be installed in the operating system without violating its integrity. The proposed solution is transparent to package managers and requires no changes in how the software packages are built and distributed. The evaluation shows that the approach is practical. It supports 99.76% of packages available in the main and community repositories of Alpine Linux while increasing the total repository size by 3.6% and incurs low performance overhead when installing software updates.

1.8 Scope and Goals

Thesis Statement

State-of-the-art security technologies, such as trusted execution environments and trusted computing techniques, protect the confidentiality and integrity of high-assurance security systems' execution and data by running them on top of a trustworthy operating system and in strong isolation from other software executing on the computer. However, these technologies were designed for different use cases, operate under various threat models, and offer distinct security guarantees. It is unclear whether their combination gives any security advantages and what possible security, performance, and usability trade-offs must be made

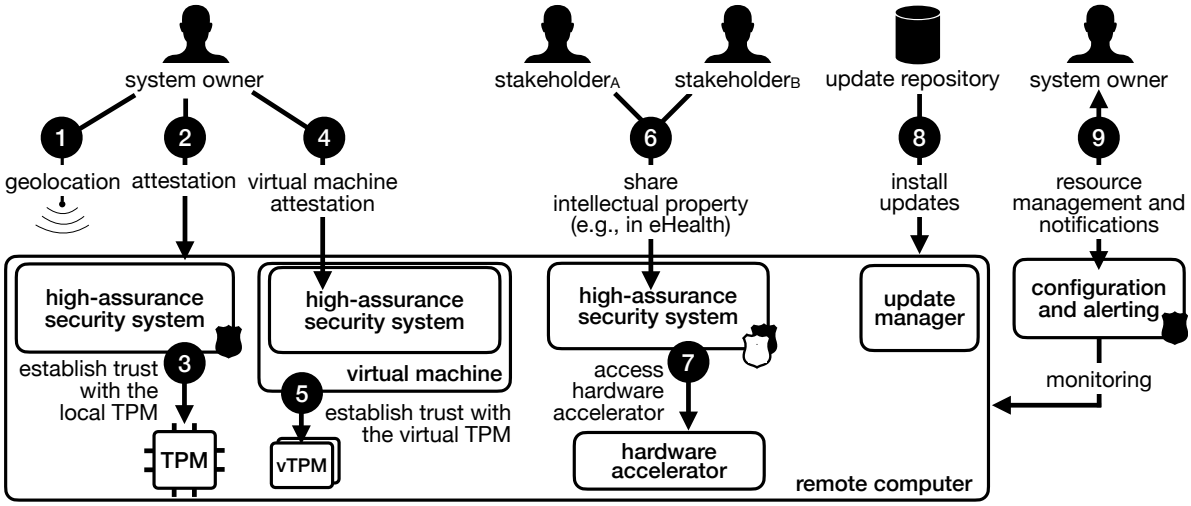


Figure 1.1. Research problems addressed in this thesis. The TPM stands for the trusted platform module, a secure element collecting integrity measurements of software executing on a computer. The vTPM stands for a virtual trusted platform module, a TPM emulator collecting virtual machine integrity measurements.

to use them together. This thesis explores how high-assurance security systems can benefit from both trusted computing and trusted execution environments in a secure, practical, and efficient way.

Scope

In this thesis, I focus on leveraging well-established state-of-the-art security techniques to solve existing problems in domains of remote attestation, secure remote computation, confidential computing, and secure multi-party computation (Figure 1.1).

I start with a fundamental problem of how to verify that a remote computer I interact with is in the expected geographical location, *i.e.*, inside a trusted data center (❶). Having a technical assurance that a computer is in the specific data center is the first step in establishing trust with that computer because it allows us to assume that the computer is protected from physical and hardware attacks. Next, I tackle the problem of verifying that this specific computer runs only the expected software in the expected configuration (❷). For that, I solve the cuckoo attack problem that prevents from establishing trust with a secure element collecting software integrity measurements (❸). After that, I address the issue of how to verify that the operating system running inside a virtual machine executes the expected software in the expected configuration (❹). For that, I analyze the existing state-of-the-art approach of virtualizing a security element compliant with the trusted platform module standard [90], and I propose how to improve it (❺).

After solving the problems ❶-❺, I show how the proposed mechanisms might be used in practice. I tackle the problem of how multiple stakeholders could cooperate to perform collaborative computation on remote computers (❻) and how they could trade-off between security and performance (❼). Specifically, how they might agree on which security mechanisms they want to rely on to protect their workloads while gaining access to hardware accelerators.

Finally, I deal with practical issues that limit the usage of integrity enforcement and moni-

toring techniques in production. First, I investigate how to safely install software updates on an integrity-enforced operating system (❽), *i.e.*, I look for a solution in which a remote verifier who monitors the integrity of the operating system can ensure that the new integrity state is a result of the trusted update and not of an attack. Second, I check how a system owner could in practice manage a group of resources, *i.e.*, define, configure, and monitor remote computers that differ in terms of running workloads and applied security mechanisms (❾).

Goals

The main goal of this thesis is to build a framework to harden high-assurance security systems. The design goals are:

- **Security.** The framework should provide strong security guarantees to high-assurance security systems. It should allow individual processes to be run in isolation from privileged software (under the trusted execution environment threat model) and on top of a trustworthy operating system (under the trusted computing threat model).
- **Attestation.** The system owner should obtain technical assurance that the high-assurance security systems execute in well-defined geographic locations inside an execution environment meeting his security requirements.
- **Practicality.** The framework must support running legacy systems without requiring source code changes. It is acceptable to instrument source code at the compilation level or run inside virtual machines. It must also support software updates and incur acceptable, low ($\leq$ 10%) performance overhead.
- **Usability.** The framework must be configurable to individual use cases by allowing users to declaratively state their trust boundaries and make a trade-off between security and performance. It should permit central management (configuration distribution and notification collection) of multiple computing resources.

Limitations

This thesis does not tackle problems of how to ensure the runtime integrity of the process code and data or how to ensure the control flow integrity of running processes. I assume the existence of corresponding methods, like [143, 180], that might be implemented in the presented solutions independently. I do not tackle the problem of how to ensure that the binary corresponds to the expected source code certified by the user as correct, a problem known as *trusted compilation*. I either consider problems of how to ensure that software is free of vulnerabilities or how to ensure the system is free of misconfigurations. For that I assume that corresponding techniques, such as fuzzing [281], formal proofs [287], unit and integration testing, code reviews, automated verification of configuration compliance with expected regulations [215], and other good programming practices are sufficient. Finally, I assume hardware implementation trustworthy, skipping the discussion on vulnerabilities of hardware-specific firmware [282], such as vulnerabilities in the Intel management engine (*e.g.*, CVE-2017-5689, CVE-2017-5705) or in the microcode implementing CPU-specific features [154].

1.9 Summary of Contributions

This thesis makes the following contributions:

(i) A protocol that verifies that the physical computer is in the expected data center (chapter 3).

(ii) A policy-based protocol that verifies the load-time and runtime integrity of the operating system (chapter 3).

(iii) A novel, practical, and formally proven cuckoo attack defense mechanism that establishes trust from the inside of the trusted execution environment to the secure element compatible with the trusted platform module standard (chapter 3).

(iv) A policy-based remote attestation protocol attesting to the virtual machine's runtime integrity (chapter 4).

(v) A method that establishes trustworthy virtual TPMs for virtual machines (chapter 4).

(vi) A multi-stakeholder machine learning framework that enables selection of a trade-off between the security and performance, and usage of hardware accelerators (chapter 5).

(vii) A practical method enabling software updates of integrity-enforced operating systems (chapter 6).

(viii) The implementation of a web-based service enabling management of multiple computing resources, management and distribution of configurations, automatic resource provisioning, and alerting (chapter 7).

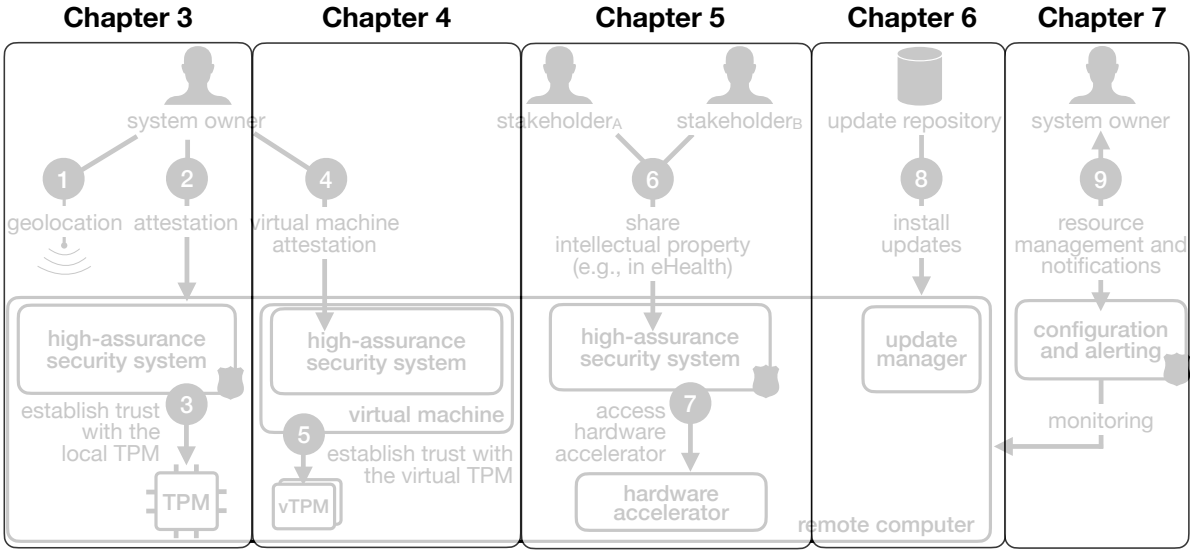


Figure 1.2. Research problems addressed in differrent chapters in this thesis.

1.10 Organization

Figure 1.2 shows how the remainder of the thesis is organized.

In chapter 2, I introduce existing concepts and technologies designed to protect computing devices and software against physical or software attacks, and techniques to obtain a cryptographical proof of software executing on a remote computer.

In chapter 3, I show the design, implementation, and evaluation of CHORS, an integrity monitoring and enforcement framework that establishes trust with a remote computer. This chapter tackles Problems ❶-❸.

In chapter 4, I discuss the design, implementation, and evaluation of TRIGLAV, a technique extending CHORS, which verifies and enforces the runtime virtual machine integrity. This chapter tackles Problems ❹-❺.

In chapter 5, I present the design, implementation, and evaluation of PERUN, a framework for selecting the trade-off between security and performance when running multi-stakeholder computations. This chapter tackles Problems ❻-❼.

In chapter 6, I show the design, implementation, and evaluation of ROD, a trusted software repository enabling software updates of integrity-enforced operating systems. This chapter tackles Problem ❽.

In chapter 7, I present the security configuration management and monitoring system that simplifies the security policy management, deployment, and monitoring of the computer's integrity. This chapter tackles Problem ❾.

I conclude the work in chapter 8.

2 Background

High-assurance security systems are deployed as multiple services distributed across multiple computers to ensure high availability, fault tolerance, and resource scalability. As such, their architects, owners, and security officers face the problem of *secure remote computation*, *i.e.*, how to ensure the confidentiality and integrity of data and code executing on a remote computer? This chapter explores existing techniques that allow users to establish trust with a remote computer, attest to the integrity of software running on such a computer, and protect the integrity and confidentiality of individual applications' code and data against malicious operating systems and administrators.

Figure 2.1 shows existing defense mechanisms used to protect computing systems at different levels. In section 2.1, I discuss general practices designed to protect computing devices against physical and hardware attacks. Next, in section 2.2, I dive into concepts that allow verification of the operating system's trustworthiness, *i.e.*, the integrity of the computer boot process and the operating system's runtime execution. Then, in section 2.3, I give a brief overview of the existing trusted execution environment technologies. I conclude in section 2.4, discussing the trusted execution technology that enables isolation of a single

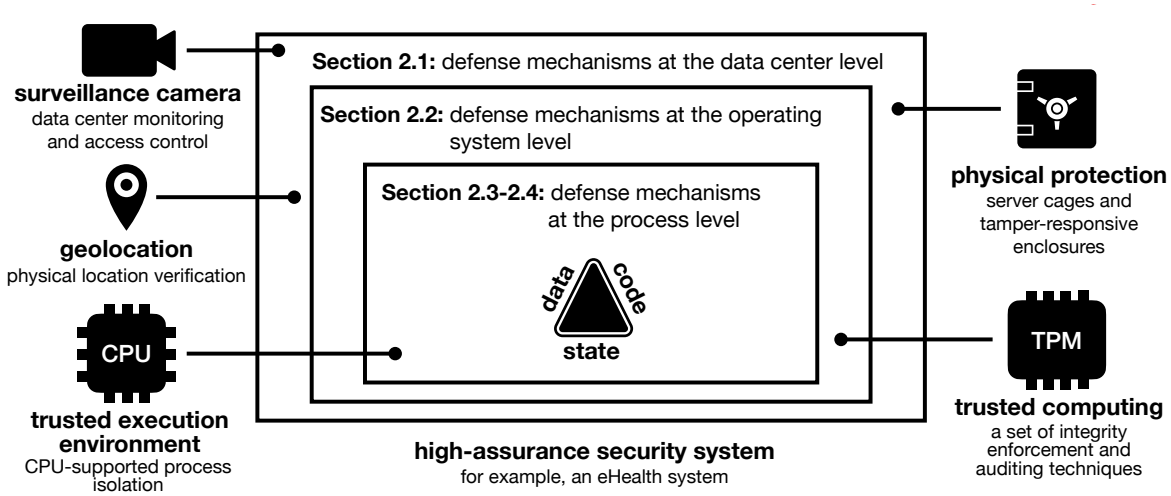


Figure 2.1. Overview of mechanisms protecting high-assurance security systems against a wide range of threats.

process from an untrusted operating system and administrator.

2.1 Physical Protection of Computing Resources

An adversary with physical access to computing resources can perform a wide range of physical and hardware attacks to violate the security guarantees offered by software and hardware mechanisms. She can mount attacks by (i) attaching malicious hardware devices to tamper with the memory content [6]; (ii) hijacking of communications on buses connecting the CPU with peripheral devices in order to manipulate data read and written by the CPU [272]; (iii) freezing the memory chip to retain secrets after the power is lost [100]; (iv) injecting faults by supplying voltage out of the CPU operational range [189] or flipping individual bits with laser beam [15] or heat [86] to bypass security checks; or (v) observing side effects of computation, like power consumption [153] or computation duration [152] to extract cryptographic keys.

The implementation of safeguards against physical and hardware attacks is difficult, costly, and therefore only performed in sophisticated devices, like hardware security modules (HSMs) used by governments, banks, or the military to guard access to cryptographic material. HSMs are not general-purpose computing devices, rather cryptographic coprocessor, sophisticated secure element, designed for very special use cases. For the sake of scalability and cost-effectiveness, general-purpose computing devices implement instead the *defense in depth* approach that restricts access to the physical machine, thus protecting from these classes of attack.

HSMs meet the highest security level, according to the federal information processing standard publication (FIPS) 140-2 level 4 or level 3 standards [120]. HSMs detect and respond to physical and hardware attacks in real-time. For example, IBM® 4769 Crypto Card [109] is enclosed in a dedicated tamper-responsive enclosure that actively detects physical and environmental attacks, such as probe penetration, side-channel attacks, power or temperature manipulation, and many more. Specifically, it erases the security-relevant material once an attack attempt is detected. Due to the high costs of production, certification, and maintenance, this level of security does not apply to general-purpose computing devices.

Instead, data center owners implement the defense in depth approach to protect computing devices from malicious actors. First, they limit access to the data center and individual server rooms to a limited number of employees. Second, computers requiring an additional level of security are locked inside electronically controlled security cages that are video monitored from inside the cages. Any maintenance work requires approval and execution of dedicated security procedures that ensure the trustworthy behavior of the operator. The use of cages is primarily intended to mitigate attacks performed by malicious insiders [42], such as malicious data center operators.

2.2 Trusted Computing Techniques

Trusted computing techniques (TCTs) offer well-established, widely available methods to build the hardware-based computer's identity, record and attest to software integrity, and prevent unauthorized changes to software configuration [93]. The crucial TCTs features are (i) remote attestation, *i.e.*, auditing, of what software has executed on the computer, and

(ii) *integrity enforcement* mechanisms ensuring that only expected software in the expected configuration can execute on the computer.

2.2.1 Security Guarantees

TCTs define how to measure, store, enforce, and report the *load-time integrity* of firmware and software that has been loaded to the computer's memory since the moment a computer was powered-up. The *reporting* capability (also referred to as *auditing* or *remote attestation*) allows verification that the operating system is in the expected, well-defined state, while the *enforcement* capability prevents the operating system from moving into an untrusted state by refusing to load an unknown, potentially malicious software to the memory. Crucially, the reporting capability verifies that the enforcement mechanism is enabled and certifies this to a remote entity with the help of the *secure element*.

Secure Element

The secure element is a cryptographic coprocessor compliant with the trusted platform module (TPM) standard [90], which defines security functionalities allowing for computer integrity auditing. The goal of having an independent [3] secure element is to make these security functionalities available for integrity measurements from the very first moment of the boot process and resist software-based attacks originating from the potentially malicious software trying to tamper with its own measurements. Consider malware taking control over the operating system. Without the tamper-resistant storage provided by the secure element, malware might vanish the proof of its existence (its integrity measurements) from the storage, thus successfully hiding its existence.

Hardware Attacks

The most popular secure element implementations are discrete TPM chips attachable to a motherboard. They are vulnerable to simple hardware attacks, however. An adversary hijacks the packages transferred via the physical bus between the TPM and the CPU because the communication is neither integrity-protected nor authenticated. This allows him to inject, modify, drop, and read arbitrary data, gaining full control over the integrity measurements stored and certified by the TPM [272, 270, 163, 142, 238, 50].

TPM chips require additional, independent protection mechanisms guaranteeing that the host computer is physically isolated from the adversary in order to maintain their promised security guarantees [25]. This is typically realized by locking servers inside security cages in an access-controlled data center (see section 2.1). Other implementations, such as the TPM functionality directly integrated into the CPU chip [121] or firmware TPMs [213], might resist some of the hardware attacks due to the physical protection offered by the CPU chip packaging or firmware isolation.

[3] The word 'independent' refers to the architectural binding of the functionality and not physical implementation or availability. Operations performed by the secure element can be processed in parallel to the code executing on the main processor.

Trusted Computing Base

The trusted computing base is a parameter defining all components responsible for providing security guarantees to the computing system. The lower the trusted computing base, the better, because there are fewer components in which vulnerability exploitation or misbehavior could lead to violation of the system's security guarantees.

TCTs have a large trusted computing base that includes all software executing on the computer, starting from firmware, bootloader, kernel, and operating system, finishing on applications running in the operating system. TCTs provide tooling to verify that the software loaded to the memory is the expected (trusted) software in the expected configuration.

TCTs define additional hardware-based mechanisms that reduce the trusted computing base size. The dynamic root of trust for measurements (DRTM) [235], also referred to in the literature as *late launch*, is a hardware CPU extension that allows a warm system reset. It dynamically creates a clean execution environment regardless of what has been executed previously, *e.g.*, firmware. I discuss this technology in more detail in subsection 2.2.4.

Load-time Integrity

TCTs are blind to changes occurring directly in the memory because they offer only load-time integrity guarantees. A load-time integrity measurement (or *integrity hash* or simply *hash*) is an output of a cryptographic hash function [210, 55] calculated over the software binary at the time it is loaded to the memory. The integrity measurement is used as a fingerprint to distinguish between legitimate (allowed, known) and untrusted (unknown, possibly malicious) software. Notably, the integrity measurement is calculated only once, at the time when the software is loaded to the memory, because then the mapping of the process memory to a deterministic hash becomes difficult.

Relying just on the load-time integrity has security implications because the legitimate software loaded to the memory can be attacked using memory corruption vulnerability exploits [195] or devices directly accessing the computer memory [6, 178]. Such attacks are not detected by TCTs because they can be executed without running malware on the same operating system. Thus, a typical assumption of TCTs is that a legitimate software loaded to the memory behaves legitimately during its entire life cycle.

Additional techniques must be used to ensure the correct behavior of legitimate software during runtime. The memory management engine (MME) [4] can transparently encrypt and decrypt the data leaving and entering the CPU chip to prevent an adversary from reading and tampering with the data stored in the main memory, such as dynamic random-access memory (DRAM) [97, 140]. The memory corruption vulnerabilities can be mitigated with the control flow integrity [143], fuzzing [281], use of dedicated compilation techniques (*e.g.*, compiling source code as position-independent executables together with stack-smashing protection), usage of memory-safe languages (*e.g.*, Rust [180]), or formally proving software implementation (*e.g.*, seL4 [148] or EverCrypt [212]). In the rest of this document, I will refer to the load-time integrity property as *integrity*, assuming that some of the countermeasures mentioned above protect the runtime integrity of processes forming the trusted computing base.

[4] Nowadays, the memory management engine (MME) is implemented inside the CPU package, as in the case of Intel and AMD CPUs.

2.2.2 Trusted Platform Module (TPM)

The TPM is the standard for security co-processors defined by the Trusted Computing Group (TCG) [256]. The newest version of the TPM specification, version 2.0 [90], was introduced in 2014 and brought new features and improvements, such as support for stronger cryptographic algorithms, internal source of time, platform reboot counter, and support for an end-to-end encrypted communication [12].

The TPM chip, further referred to simply as the *TPM*, is a passive component that cannot initialize communication with any external devices or perform any action without being requested, *i.e.*, it is a coprocessor responding to the commands send by the processor. This design implies the existence of a *measuring agent*, a piece of software running on the computer that performs the integrity measurements and sends them to the TPM. The measuring agent changes due to the sequential nature of the computer boot process depicted in Figure 2.2. The consecutive firmware and software layers take control of the computer, becoming new measuring agents. The first agent, called the *root of trust*, is the first immutable piece of code initializing the computer boot procedure. It is loaded from the read-only memory (ROM) or is embedded directly in the CPU and must be explicitly trusted [5]. It initializes the *chain of trust* that allows the trust to be extended to consecutive measuring agents with the help of dedicated tamper-resistant memory, called platform configuration register (PCR), provided by the TPM.

Platform Configuration Registers

The TPM 2.0 chip has a built-in protected memory which consists of non-volatile random-access memory (NVRAM) and platform configuration registers (PCRs). PCRs are tamper-resistant and are used to store integrity measurements of firmware and software that has been executed on the computer.

PCRs are divided into *static* and *dynamic* PCRs. Static PCRs can be initialized only with the restart of the computer. Dynamic PCRs can be initialized and extended during the warm system reset only by a trusted firmware executed in a certain *locality* [132], as defined by the DRTM specification [235]. A PCR cannot be set to any arbitrary value, except for the initial value set during the PCR initialization. Then, the PCR value can only be extended with a new value, as expressed in Equation 2.1. The *PCR_extension* function implements the cryptographic hash function, denoted as *hash*, to provide the tamper-resistant property.

$$PCR\_extend = hash(PCR\_old\_value \,||\, data\_to\_extend) \tag{2.1}$$

Remote Attestation Protocol

The TPM implements the *TPM attestation protocol* [91] defining how to read and certify the PCR values to a remote entity. Specifically, the TPM issues a digitally signed report (*quote*) that certifies the integrity measurements extended to PCRs by measuring agents, using a signing key embedded in the TPM.

[5] There are methods, like Intel's Boot Guard [275], that ensure the integrity and authenticity of the code loaded from the ROM. The code stored in ROM is digitally signed by the manufacturer and the CPU verifies the signature with the key burned into the chip's e-fuses. This prevents bootstrapping the computer with untrusted code.

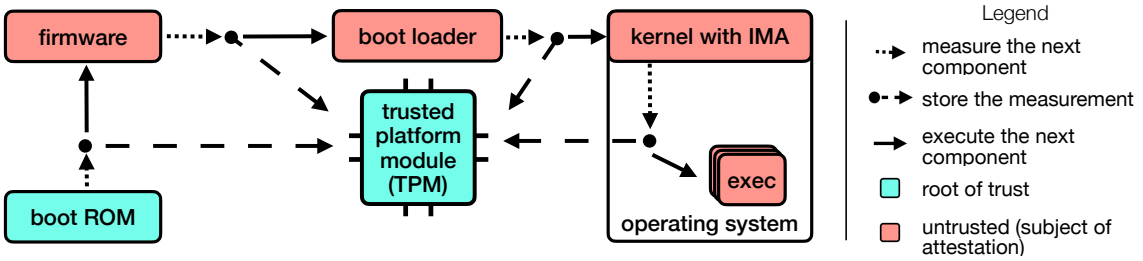


Figure 2.2. The chain of trust: Each boot component measures the integrity of the next boot component before executing it. The measurements are stored inside the TPM chip. The *boot ROM*, which is the first boot component, initializes the chain of trust. It is immutable and must be explicitly trusted.

The signing key is an asymmetric cryptographic key embedded in the TPM chip at the manufacturing time in a way it is only known to the TPM. The TPM also stores a digital certificate containing the public key corresponding to this signing key. The certificate is signed by a manufacturer or the computer owner. Consequently, it is possible to check that a genuine TPM chip produced the quote because the quote's signature is verifiable using the public key read from the certificate linked to a trusted entity.

2.2.3 Secure Boot and Measured Boot

The *secure boot* [269] (also known as *verified boot*) is the state-of-the-art technology enforcing that only trusted software bootstraps the computer. It follows the *chain of trust* concept (Figure 2.2) where each *boot component* (*i.e.*, firmware, bootloader) calculates an integrity measurement (a cryptographic hash) of the next boot component and executes this component only if its hash matches a corresponding digital signature issued by a trusted entity. I say that the secure boot enforces the boot integrity because the boot component aborts the boot process when it fails verifying the signature of the next boot component. In other words, the secure boot process guarantees that the booted system has correct load-time integrity, assuming lack of physical or hardware attacks.

The *measured boot* [254, 255] (also known as the *trusted boot*) complements the secure boot by enabling auditing of the boot process. The consecutive boot components extend hashes to TPM's PCRs. The TPM then vouches for the load-time integrity state of the established execution environment by certifying PCRs values. Like this, a verifier gets a technical assurance that indeed the expected boot components bootstrapped the computer. Please note here that when the measured boot is not used with the secure boot, it is possible to load the system which load-time integrity is not valid. It is the responsibility of the remote verifier to attest to the load-time integrity of the booted system before establishing trust with it.

2.2.4 Dynamic Root of Trust For Measurement

TCTs define a technology, called dynamic root of trust for measurements (DRTM) [235], that allows for the establishment of a new, clean execution environment at an arbitrary point in time without the hard reset of the computer. A CPU implementing this technology halts the execution of all cores except one, which then runs a vendor-provided trusted firmware. This firmware resets a dedicated set of PCRs to their initial values and extends them with the measurement of the piece of code requested to be loaded and executed. The tboot project [127]

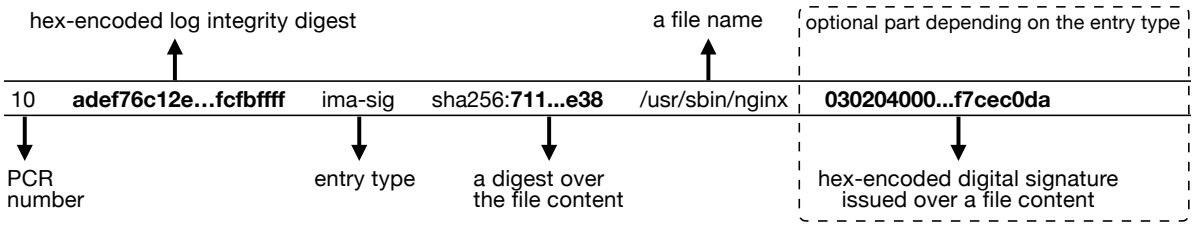


Figure 2.3. IMA log entry format. The *PCR number* indicates to which TPM's PCR the kernel extended the hash calculated over the entry content. Depending on the type of the entry, IMA log can store additional data related to the measured file, such as the digital signature stored in the file's extended attributes. The *hex-encoded hash over the entry* is a hash calculated over the entire entry appended to the IMA log, allowing for detection of tampering with the entry's content. The *hex-encoded hash over the file content* is the hash calculated by the kernel before loading the file to the memory.

is an example technology leveraging DRTM. It is a bootloader that securely measures, loads, and executes the Linux kernel and the minimalistic root filesystem (*initramfs*), regardless of the trustworthiness of the boot components used to bootstrap the computer.

DRTM implements protection against software-based attacks by ensuring that the piece of code requested to execute has exclusive control over the computer. When a privileged software requests launch of a piece of code in a DRTM, it invokes a dedicated CPU instruction (SKINIT for AMD-based CPU or SENTER for Intel-based CPU), providing a memory address where a piece of code resides. The trusted DRTM firmware (a CPU's microcode implementing the DRTM launch) disables direct memory access (DMA), interrupts, and debug capabilities. It halts all CPU cores except the main one [45, 184] that will execute the requested piece of code. It also initializes dynamic PCRs, a dedicated set of PCRs storing DRTM measurements, to which it sends the integrity measurements of the piece of code to be executed. Eventually, this piece of code executes in a clean environment with full control over hardware.

DRTM relies on the TPM to store integrity measurements of the measured execution environment and to report them to a remote party interested in the proof of the load-time integrity of the code executed in DRTM. The TPM supports DRTM with a custom set of registers called *dynamic PCRs*. These registers extend the standard PCR functionality with authorization of who and when can reset or extend the dynamic PCR. Specifically, the dynamic PCRs can be reset in runtime only by the trusted DRTM firmware.

The DRTM is implemented in Intel and AMD commodity CPUs under the names Intel trusted execution technology (TXT) [87] and AMD secure virtual machine (SVM) [3], respectively. Nowadays, DRTM is utilized by the cloud management software [128] to securely load a hypervisor irrespective of the trustworthiness of the firmware [127] or, as demonstrated by researchers, to run a single application in isolation from firmware and operating system [184].

2.2.5 Operating System's Runtime Integrity Measurement and Enforcement

The Linux integrity measurement architecture (IMA) [225] is the implementation of the integrity measurement architecture [89] proposed by the Trusted Computing Group [256]. IMA extends the functionality of the measured and secure boot to the operating system level. Specifically, IMA, which forms part of the kernel, measures and, optionally, enforces files' integrity before they are loaded to the memory during the operating system's runtime. It also integrates with the TPM for auditing purposes.

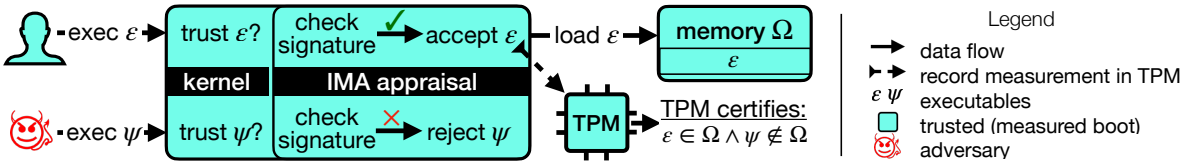


Figure 2.4. Integrity measurement architecture (IMA) is part of the kernel. It approves software to execute and provides reporting functionality to verify what software has been executed since the load of the kernel.

Auditing

The auditing is realized by maintaining a tamper-proof file called *IMA log* storing integrity measurements of all files loaded to the memory since the load of the kernel. This file, the integrity of which is certified by the TPM, can be transferred to an external entity as proof that the kernel launched only expected software, *i.e.*, executable, configuration files, dynamic libraries. The tamper-proof property of the IMA log is maintained using a dedicated format where a hash of each new entry is extended to the PCR, as depicted in Figure 2.3. Each entry in the IMA log represents a single integrity measurement corresponding to a file (a configuration file, executable, or dynamic library) loaded by the kernel to the memory. The integrity of the IMA log file is verifiable by recalculating the integrity hashes over consecutive entries and comparing the result with the PCR value certified by the TPM. Any tampering with the IMA log, such as modifying the entry content, adding, removing, or reordering entries, is detected.

Integrity enforcement

The IMA implementation in the Linux kernel comes with a built-in integrity enforcement mechanism called IMA-appraisal [101]. It ensures that the kernel loads only software whose integrity is certified with a digital signature. Figure 2.4 shows how the mechanism works. IMA reads the digital signature corresponding to the given executable from the file system and verifies that the cryptographic hash over the executable (*integrity measurement*) matches the original integrity measurement signed by the trusted party. Typically, the owner of the running operating system digitally signs only these files which he trusts to be correctly implemented software and configuration. For example, he would typically trust files that: (i) originate from trusted places like the official Linux git repository, (ii) pass security analysis like fuzzing [281], and (iii) were generated using compilation techniques preventing the exploitation of memory vulnerabilities. Such techniques include compiling source code as position-independent executables together with stack-smashing protection as done with packages of the Alpine Linux [7].

2.2.6 TPM Alternatives to Boot Code Integrity Protection

Although the TPM standard is widely available in server and desktop computers, it suffers from limitations that led to the development of alternative technologies used to protect the boot integrity of computers in cloud data centers.

The first problem (P1) is that the TPM is a passive device and, as such, cannot verify the first boot code integrity, so-called *the core root of trust for measurements*. Consequently, an

adversary who successfully attacks the supply chain [220] or has physical access to the computer gains control on the first boot code initializing the computer. Like this, he can mitigate the secure boot process to load an arbitrary, vulnerable operating system. The TPM does not help to detect the attack because it explicitly trusts the first piece of code, *i.e.*, the first measuring agent, starting the measured boot.

A second problem (P2) with the TPM is that it is vulnerable to simple hardware attacks [272, 270, 163, 142] because a discrete TPM chip communicates with the CPU via a communication bus accessible to an adversary with physical access to the computer. Consequently, an adversary can tamper with the TPM by resetting it and replaying arbitrary measurements, overcoming the trusted boot protection used by BitLocker or Linux unified key setup (LUKS) [28].

Cloud providers addressed these problems by introducing dedicated hardware-based protection integrated directly in the processor (Microsoft Pluton) or as discrete hardware (Google Titan, Amazon Nitro).

Microsoft Pluton

Microsoft Pluton is an intellectual property (IP) security subsystem integrable with system on chip (SoC) [241]. It provides hardware security features, such as hardware root of trust, random number generator, cryptographic functions implementation and their accelerators, system identity, and hardware-based attestation. The Pluton addresses P1 because it is the first processor coming out of reset that initializes other SoC components after it successfully boots with the boot code loaded from the *on-chip* ROM. Internally, Pluton offers secure storage for integrity measurements and cryptographic keys to remotely attest to the measured software's integrity – a functionality similar to the TPM. Because Pluton is directly integrated into the SoC, it prevents simple hardware attacks, addressing P2. Moreover, Pluton emulates the TPM allowing computers not equipped with the TPM to leverage Pluton directly. Pluton is used to protect Microsoft Windows personal computers [268], the Azure cloud, and IoT devices [241].

Google Titan

Google protects the boot integrity of computers by building the Google Cloud Platform with a dedicated, purpose-built chip called Titan [227]. Titan's main goal is to ensure the integrity and authenticity of the first boot code loaded on the computer. To achieve this, it interposes the communication between the CPU and the flash memory containing the boot code. It verifies the authenticity and integrity of the boot code using public-key cryptography. Only when the boot code is valid, Titan allows the rest of the machine to come out of reset. Titan addresses P1 but does not address P2. This is because Titan focuses on usability: It can be easily integrated with existing CPUs, although it lacks protection against hardware attacks. This is a reasonable approach because Titan is a proprietary solution dedicated to protect the Google's data center from software-level attacks or attacks from peripheral devices, while hardware attacks are mitigated with access control mechanisms at the data center level.

Amazon Nitro

The Amazon cloud builds on the Amazon Nitro architecture, which decomposes the hypervisor functionality into i) hardware-assisted services implemented on dedicated peripheral component interconnect express (PCIe) cards, ii) a small software hypervisor performing

memory management, CPU scheduling, error handling, and iii) a security chip providing hardware root of trust.

The security chip is integrated into the motherboard and traps all communication to non-volatile memory [102]. It prevents arbitrary changes to the memory storing the boot code and only allows updates originated via the Nitro PCIe card. The security chip supports secure boot and measures the integrity of firmware, comparing it with the whitelist measurements stored in the security chip [102]. The security chip addresses P1. Due to the lack of publicly available technical details, it is hard to reason what security guarantees are offered against hardware attacks.

2.3 Trusted Execution Environment

Trusted execution environment (TEE) is a mechanism that aims at creating dynamically disjoined isolated execution environments, commonly referred to as *enclaves*, on the same computing resources. Unlike the process isolation mechanism provided by the operating system or virtual machine isolation mechanisms provided by the hypervisor and hardware, TEE promises strong confidentiality and integrity guarantees to an application executing inside the enclave in the face of the untrusted operating system, hypervisor, system administrator, and peripheral devices with direct memory access.

Different TEE implementations exist. They differ in terms of offered security guarantees, the trusted computing base (TCB) size, performance, and the presence of certain security features, like remote attestation. In general, we can partition existing TEEs into ones that isolate a single process and the ones that isolate the entire operating system executing in a virtual machine.

The most known representative of the first group is Intel software guard extensions (SGX), a hardware-supported TEE mechanism present in modern Intel CPUs. This is also the TEE mechanism that I describe in more detail in this chapter because it is the TEE on which I heavily rely in the rest of our work. Other TEEs that fall into this category are TIMBER-V [265], Sanctum [46], MultiZone [104].

On the other spectrum, AMD secure encrypted virtualization (SEV) [116], Intel trust domain extensions (TDX) [123], and IBM protected execution facility (PEF) [105] all allow for a complete virtual machine to be run inside an enclave. Their main advantage is that they transparently support running legacy applications in virtual machines requiring zero source code changes while protecting against the hypervisor and its operator. Compared to TEEs isolating a single process, they have much higher TCB because the entire operating system running inside the virtual machine must be trusted. Consequently, solutions relying on these TEEs require TCTs mechanisms to enable the auditing and enforcement mechanisms of the runtime operating system integrity.

2.4 Intel SGX

Intel software guard extensions (SGX) [45, 185] is a TEE mechanism implemented on Intel CPUs, starting from the Skylake microarchitecture introduced in 2015. It permits the execution of a single process inside an enclave. Still, it requires an operating system to maintain the enclave's lifecycle and manage the computing resources shared with other processes and peripheral devices. SGX operates under a threat model where supervisor software, *i.e.*,

firmware, operating system, as well as a system administrator with physical and root access to the computer and peripheral devices are untrusted.

More formally, Intel SGX is an extension of the x86_64 architecture. It introduces new instructions required to command the CPU to manage the enclave, *i.e.*, allocating a dedicated protected memory region, copying initial enclave code to that memory region, measuring the initial code, switching context to and from the enclave, sealing data, paging, and generating a cryptographical proof of the enclave's identity for remote attestation.

Virtually, an SGX enclave execution is similar to regular process execution. The enclave execution thread is interruptible and preemptible, allowing the operating system to retain control over the CPU time scheduling and resource allocation, including enclave creation, destruction, and memory swapping. The Intel SGX design ensures that the enclave code and data are isolated from other software, including during the enclave execution. The CPU guards access to the protected enclave memory, allowing only the enclave thread executed by the logical CPU in a dedicated CPU mode, called *enclave mode*, to access the protected memory regions containing the enclave's own code and data.

2.4.1 Security Guarantees

SGX protects both confidentiality and integrity of the application's code and data against software attacks launched from privileged software and other applications executing on the same computing resources. The data confidentiality, integrity, and freshness are guaranteed during runtime and at rest when the data resides in untrusted memory, such as DRAM or a hard drive. In such situations, the CPU cryptographically protects the data before it leaves the CPU package. SGX detects tampering with the enclave's memory and prevents it or aborts the enclave's execution [185] once the tampering is detected. It is the enclave's responsibility to properly sanitize untrusted input received from the operating system to prevent Iago attacks [39, 48].

Trusted Computing Base

The SGX design aims to minimize the TCB size. The application owner must trust the Intel CPU's, the SGX implementation, and that his own application implementation is free of memory-corruption vulnerabilities. The kernel, user space applications, system management mode (SMM), virtual machine monitor, and any other software and hardware are considered untrusted. However, the operating system is essential from the enclave's point of view. It controls the enclave's lifecycle and access to computing resources and external devices. Thus, the enclave's availability, which is out of the scope of the Intel SGX threat model, depends on the trustworthiness of the operating system and system operator.

Memory Protection

SGX defines a dedicated memory region, called processor reserved memory, to which the CPU guards access. Inside this memory region, the CPU implements the enclave page cache (EPC) which stores the enclave's code and data as well as SGX-specific structures [185]. Crucially, only enclaves can access this memory region; CPU denies access attempts from any other software, even the most privileged ones on the x86 architecture, like SMM, and from peripheral devices. Because the size of the EPC memory is limited, the operating system can

evict EPC pages to the untrusted memory, like DRAM. Even in this situation, SGX maintains the confidentiality, integrity, and freshness guarantees because the pages leaving the EPC memory are encrypted by the CPU, and only this specific CPU can decrypt them when they are moved back from the untrusted memory into the EPC. The memory encryption engine, which is part of the CPU chip, implements the SGX memory protection mechanism.

Load-time and Runtime Integrity

SGX measures the application's load-time integrity and ensures that the enclave's memory cannot be modified by hardware or software outside the enclave's trust boundary [185]. However, SGX does not protect against flows in the application's implementation. An adversary can exploit memory corruption vulnerabilities leading to control flow hijacking or tampering with the enclave's memory. In such a case, the SGX will attest to the enclave's integrity with the enclave's load-time integrity measurement that does not correspond to the enclave's runtime integrity. These attacks do not break the SGX threat model because it is the application's owner's responsibility to protect the application's implementation with techniques described in subsubsection 2.2.1. This is the same assumption as in TCTs, and most of the already mentioned mitigation strategies can be used to protect the enclave's code.

Side-channel and Hardware Attacks

SGX is vulnerable to side-channel and microarchitectural attacks [260, 231, 261, 262, 189, 237] that violate the SGX confidentiality guarantees. A malicious application sharing the same computing resources (CPU caches, CPU cores) as the victim application can learn some information by exploiting side-channels, like cache access latency or transient execution. The microarchitectural attack allows the data in caches to be speculatively accessed before the CPU determines lack of permissions [260].

2.4.2 Enclave Initialization and Execution

The operating system initializes the enclave by requesting the CPU to copy the enclave code and data from untrusted memory to the EPC pages. Once all enclave pages are copied, the enclave initialization is finished. The CPU disables any further ability to add new EPC pages and measures the application's integrity. The resulting hash is later used to certify the load-time integrity of software executing inside the enclave.

The operating system controls the enclave's lifecycle. It executes a dedicated CPU instruction to switch the context to a protected mode in which the control flow is switched to the enclave code and state. It can interrupt the enclave's execution at any time, causing context switch from the enclave mode back to the userspace and kernel mode. With the help of additional instruction, the operating system can evict the enclave's memory pages to the disk, restart the enclave execution, or destroy it.

2.4.3 Remote Attestation

The *SGX attestation* is a protocol in which another software or application's owner (verifier) ensures that the application runs inside an enclave on the SGX enabled platform. The ap-

plication running inside the enclave generates a report using secure hardware, an Intel CPU with SGX extension, which the verifier uses as proof that the expected application executes inside the enclave. The SGX local attestation defines a procedure in which one application ensures that the application runs inside an enclave on the same CPU and has a specific enclave hash measurement (MRENCLAVE). Similarly, SGX remote attestation is a protocol where one application learns that another application with specific MRENCLAVE runs inside an enclave on a different genuine Intel CPU.

During the enclave initialization, the CPU cryptographically hashes the enclave code and data copied to EPC pages to obtain the MRENCLAVE. Once the initialization process is finished, the CPU calculates the final hash representing the initial enclave state loaded to the EPC memory. This hash is used later during the attestation to prove to another entity the load-time integrity and identity of the application executing inside the enclave.

At a high level, the SGX remote attestation involves three parties: (i) the remote verifier willing to establish trust with his application executing in an enclave, (ii) the to-be-attested application executing inside an enclave, and (iii) a privileged *quoting enclave* implemented by Intel that signs the attestation report. First, the remote verifier sends a challenge to the application, *i.e.*, a unique random nonce that is used to ensure the liveness. Second, the application executing inside the enclave generates a manifest that includes this challenge and an individually generated public key. Third, it passes the hash over the manifest to the CPU, which knows the identity of the enclave, and generates the attestation report that includes the enclave's hash and the hash of the manifest. Finally, the application forwards the report to the quoting enclave, a privileged enclave implemented by Intel that executes on the same CPU and has access to the attestation key. The quoting enclave verifies that the report belongs to an enclave executing on the same CPU and signs the report using an attestation key. Eventually, the application sends back the report to the remote verifier, who validates the report's signature and integrity and finally compares the challenge to ensure freshness. Once complete, the verifier can establish secure communication with the enclave because it has the enclave's public key, and only the enclave knows the corresponding private key cryptographically protected by the CPU.

Each Intel CPU has unique secrets fused in one-time programmable memory during the manufacturing time. These secrets are indirectly used to obtain an attestation key from the remote Intel's provisioning service, and the attestation key is then used to sign the attestation report. In more detail, Intel provides a dedicated privileged enclave, called *provisioning enclave*, that retrieves the *provisioning key* derived from the CPU secrets. This enclave identifies itself with the help of the provisioning key to a remote Intel service, called Intel attestation service (IAS) [133], and receives back the attestation key. IAS can verify that it communicates with the legitimate provisioning enclave because (i) only the provisioning enclave has access to the provisioning key derived from the hardware-based provisioning key derivation process, and (ii) Intel stores part of the CPU secrets inside a database allowing derivation of the same provisioning key. The provisioning enclave shares the attestation key with the quoting enclave using dedicated sealing keys, which allow the migration of secrets between enclaves. For more details, I refer the reader to the official Intel documentation [133, 129] and research papers [45, 9].

2.4.4 Sealing

SGX offers a *sealing* [9] property that permits confidential data to be stored in the untrusted memory, *e.g.*, a hard drive, while maintaining confidentiality and integrity guarantees. The

SGX sealing operation encrypts and signs the data leaving the enclave using a cryptographic key that is specific to the enclave and the CPU. Thus, only the same enclave running on the same CPU can read the data.

3 High-assurance Security Systems Integrity Monitoring and Enforcement

3.1 Problem Statement

High-assurance security systems [76, 58, 159] leverage trusted execution environments (TEEs) [45, 169, 172] because TEEs offer strong integrity and confidentiality guarantees in the face of untrusted privileged software, *i.e.*, firmware, hypervisors, operating system, and administrators. However, applications executing in a TEE cannot exist without the privileged software (operating system or hypervisor) that manages the computing resources and controls applications' life cycles. Thus, a *trustworthy operating system* is an essential element of each high-assurance security system because it guarantees its safety and security. Otherwise, an untrustworthy operating system might run malware that halts the victim application or steals secrets from the TEE via side-channel attacks [260, 277], as depicted in Figure 3.1. Germany introduced regulations requiring high-assurance security systems in the eHealth domain [76] to execute inside TEE on a trustworthy operating system [77]. State-of-the-art mechanisms to attest to the operating system's trustworthiness rely on the trusted platform module (TPM) [90], a secure element storing and certifying integrity measurements of firmware and operating system. Unfortunately, the TPM is vulnerable to the *cuckoo attack* (a.k.a *relay attack*) [205, 53] that makes the TPM attestation untrustworthy. We propose a novel defense mechanism against the TPM cuckoo attack, and we implement it as part of the framework responding to the German eHealth systems regulations [77].

The integrity measurement architecture (IMA) [89] and the dynamic root of trust for measurements (DRTM) [235] are state-of-the-art mechanisms providing operating system integrity auditing and enforcement. The DRTM securely loads the kernel to the memory, and IMA, which is part of that kernel, ensures that the kernel loads only software whose integrity is certified with a digital signature. Both technologies, when used together, ensure the *load-time integrity* of the kernel and software loaded to the memory during the operating system runtime. Specifically, the DRTM, a hardware technology implemented in the CPU, stops all cores except one, disables interrupts, measures the to-be-loaded kernel, and executes the kernel with the IMA integrity enforcement mechanism. IMA restricts software loaded to the

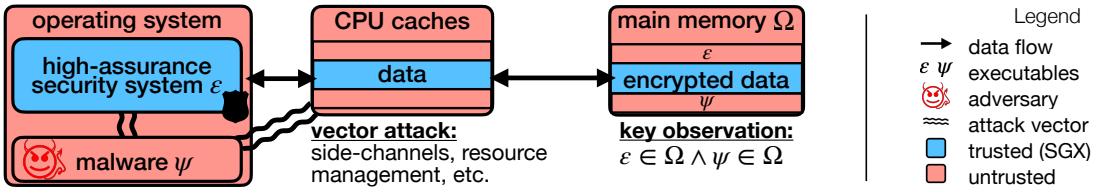


Figure 3.1. An adversary must run arbitrary software to mount a software side-channel attack that can compromise the confidentiality guarantee of Intel SGX.

memory by reading the digital signature corresponding to the given software from the file system and verifying that this software's integrity measurement (a cryptographic hash over its binary) matches the original integrity measurement signed by a trusted party (Figure 2.4). Thus, only software certified by a trusted party can be loaded to the memory by the kernel.

The TPM enables auditing of the kernel and software integrity because DRTM and IMA store corresponding integrity measurements in the tamper-proof TPM memory. The TPM then certifies the stored measurements to a verifier accordingly with the *TPM remote attestation* protocol. However, the TPM remote attestation is prone to the cuckoo attack, which is a security issue for TPM-based systems [80, 155, 38] [6]. In this attack, an adversary certifies the software integrity of the underlying computer using certified measurements of another computer (see Figure 3.2). A *verifier* connects to the compromised computer and communicates with the TPM to check the computer software integrity (❶). The adversary prevents the verifier from accessing the local TPM by redirecting communication to a remote TPM (❷). Consequently, the verifier reads the remote TPM, which attests to an arbitrary, trustworthy state (❸), not the state of the compromised computer accessed by the verifier.

The existing defenses against the cuckoo attack have limited application in real-world data centers (DCs). The first approach relies on the time side-channel [67, 234] in which a remote TPM is unmasked by observing increased communication latency. This approach requires calculation of hardware-specific statistics, is prone to false positives because the high TPM communication latency (including signature generation) makes the distance bounding infeasible [205, 155], and requires stable measurement conditions in which extraneous operating system services are suspended during the TPM communication [67] — impractical assumptions for real-world DCs. Flicker [184] adapts another approach. It exploits DRTM to run an application in isolation from the untrusted operating system, allowing it to communicate with the TPM directly. Flicker is insufficient for the targeted systems like [76] because i) it does not attest to the computer location, making the DRTM attestation untrustworthy because of simple hardware attacks [272] and cold-boot attacks [100] and ii) while it permits to split applications in multiple services that run isolated, it does not support systems with moderate throughput and latency requirements. In more detail, DRTM provides isolation in which the entire CPU executes *only a single service at a time* and a single context-switching takes 10-100s of milliseconds [184, 183]. It results in an estimated program execution's throughput of about 1-10 requests per computer per second when running multiple eHealth services, like [77]. A practical solution requires that hundreds of services are processed in parallel per computer. We require an improvement of at least one order of magnitude in throughput compared to Flicker. Other approaches [51, 52] fall short in the context of the TPM because i) the TPM is a passive device controlled by software that could counterfeit its communica-

[6] Please note that this attack is also valid for integrated TPMs and firmware TPMs because the communication to the TPM is still routed via untrusted code

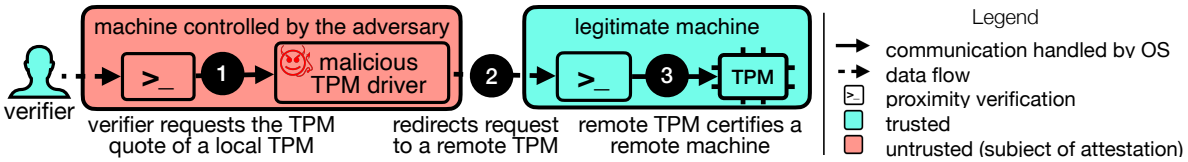


Figure 3.2. The cuckoo attack. The verifier connects to the compromised machine (left) and reads the TPM quote to verify its integrity. The quote is, however, retrieved from the remote TPM attached to a legitimate machine (right). The verifier cannot distinguish if the quote comes from the TPM attached to the local or remote machine.

tion with external devices and ii) they would require human interaction during each computer boot.

The limitations of the existing solutions motivate us to propose a new automatic, practical at the data center-scale defense mechanism that deterministically detects the cuckoo attack and allows for the processing of parallel requests. We demonstrate that despite the differences in their threat models and designs, TEE and TPM-based techniques complement each other, allowing for mitigating the cuckoo attack. Consequently, high-assurance security systems executing inside TEE can attest to the operating system integrity. Our solution builds trust in a remote computer starting from a piece of code executing inside the TEE, and then systematically extend it to the entire operating system. First, we leverage TEE to settle a trusted piece of code on an untrusted remote computer. We use it to verify that the computer is in the correct DC and mitigate the cuckoo attack. This allows us to extend trust to the TPM, then to the loaded kernel and its integrity-enforcement mechanism and, finally, to software being executed during the operating system runtime.

We implement this approach in an integrity monitoring and enforcement framework called CHORS [7], which ensures that high-assurance security applications execute on correctly initialized and integrity-enforced operating system located in the expected DC. The high-assurance security systems conform to the TEE threat model, while they gain operating system integrity guarantees under a less rigorous threat model typical for TPM-based systems. We perform security risk analysis related to the use of these techniques in §3.7.

3.2 Contribution

In this chapter, we make the following contributions: (i) We designed and implemented an integrity monitoring and enforcement framework called CHORS that i) attests to the operating system trustworthiness (§3.1,§3.4), ii) defends against the cuckoo attack (§3.6.1, §3.6.2), iii) provides a reliable approach to estimate the geolocation of physical servers beyond the simple TPM geo-tagging (§3.5.3), iv) provides local attestation, allowing decentralization of the monitoring system (§3.5.1, §3.5.4), v) the service itself can be remotely attested (§3.6.4), vi) verifies the compliance of provisioned resources with a given policy (§3.5.2, §3.5.4). (ii) We assessed the security risk of CHORS (§3.7). (iii) We demonstrated CHORS protecting a real-world application in the eHealth domain (§3.8.1). (iv) We evaluated its security and performance (§3.8).

[7] In the Slavic mythology, Chors is a Slavic god of sun, sometimes interpreted as a moon god [79].

3.3 Threat Model

We adopt the threat model of organizations, such as governments, banks, and health, legally bound to protect the security-sensitive data they process. In particular, we assume they execute high-assurance security systems in their own DCs or in the hybrid cloud in which security-critical resources are provisioned on-premises. This implies limited and well-controlled access to DCs, allowing us to assume that an adversary, *e.g.*, a rogue operator, cannot perform physical or hardware attacks. To ensure that a high-assurance security system executes inside the DC, we only presume that dedicated computers, called *trusted beacons*, are located inside that DC and cannot be physically moved outside (§3.5.3).

Initially, we only trust the CPU (including its hardware features TEE and DRTM) and a small piece of code (the *agent*). Using the TEE attestation protocol, we ensure that the legitimate agent executes inside the TEE on a genuine CPU on some computer. Then, we use the agent to verify that the computer is located in the correct DC by measuring the proximity to the trusted beacon via a round-trip time distance-bounding protocol. Once we ensure that the agent runs in the expected DC, we use it to establish trust with the local TPM with the help of our protocol formally proved to be resistant to the cuckoo attack [203]. At this point, we use the TPM to extend the trust to the kernel and its built-in integrity-enforcement mechanism, IMA. Eventually, we use IMA to expand trust to the software loaded during the operating system runtime.

High-assurance security systems executing inside the TEE follow the TEE threat model, *i.e.*, operating system, firmware, other software, and system administrator are untrusted. The additional guarantees of the operating system integrity follow the threat model of TPM-based systems, *i.e.*, software whose integrity is enforced at load-time behaves in a trustworthy way also during its execution. The runtime integrity of the process can be enforced using existing techniques, such as control-flow integrity enforcement [143], fuzzing [281], formal proofs [287], memory-safe languages [180], or memory corruption mitigation techniques (position-independent executables, stack-smashing protection, relocation read-only techniques). Please note that many of these techniques are applied nowadays by default during the software packaging process, as in the case of Alpine Linux [7].

We assume a financially or governmentally motivated adversary who might gain root access to selected computers inside a DC by exploiting network or operating system misconfigurations, exploiting vulnerabilities in the operating system, or using social engineering. Her goal is to extract security-sensitive or privacy-sensitive data, *e.g.*, personal data, credentials, or cryptographic material. She can stop or halt individual computers or processes, but she cannot stop all central monitoring service instances responsible for reporting security incidents.

We consider an untrusted network where an adversary can view, inject, drop, and alter messages. She can call the API with any parameters and configure the routing, forcing packages to choose faster or slower routes. Our network model is consistent with the classic Dolev-Yao adversary model [56]. We rely on the soundness of the employed cryptographic primitives used within software and hardware components.

3.4 Design Decisions

Our objective is to provide a design that:

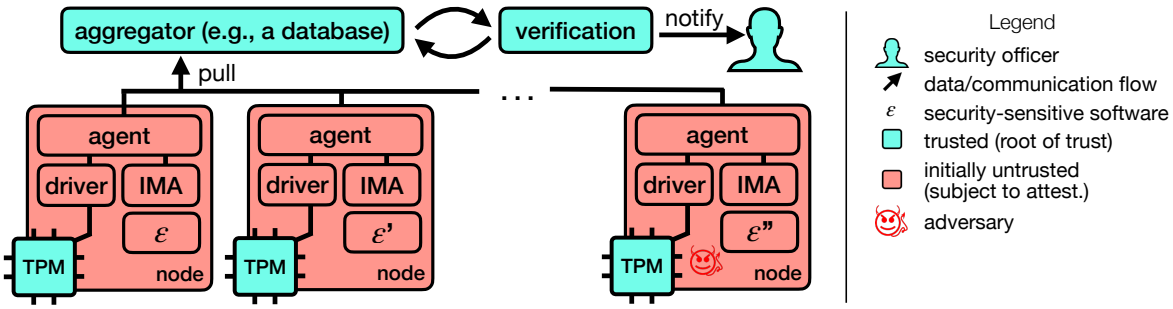


Figure 3.3. The architecture of existing integrity monitoring systems. The security officer uses a monitoring system to verify that high-assurance security systems execute on hosts running trusted software.

- enforces that only trusted software is executed on a computer;
- monitors the remote computer operating system to verify compliance to integrity requirements;
- allows high-assurance security systems to get insights into the operating system integrity.

We start by introducing the existing integrity monitoring systems architecture [125, 111, 128] and adjust it to meet the security guarantees required by high-assurance security systems. Figure 3.3 shows the integrity monitoring architecture where a central server pulls integrity measurements from computers by communicating with dedicated software, *the agent*. The agent on each computer collects data from the underlying security and auditing subsystems that measure and enforce the operating system's integrity. Central servers aggregate the data in databases, verify it against whitelists, and notify the *security officer* about integrity violations. Such architecture relies on the TPM as a root of trust.

Enforce the load-time integrity with secure boot and operating system integrity enforcement.

Secure boot [269] is the state-of-the-art technology to enforce that only trusted software bootstraps a computer. It relies on the *chain of trust* where each component measures the integrity (calculates a cryptographic hash) of the next component and executes it only if the hash matches a corresponding digital signature. The *measured boot* [254, 255] complements it by storing hashes in the TPM, thus enabling auditing.

The integrity measurement architecture (IMA) [225, 89] extends the functionality of measured boot and secure boot to the operating system level. IMA is part of the kernel and verifies all files' integrity (*i.e.*, executables, configuration files, dynamic libraries) before they are loaded to the memory. In particular, IMA-appraisal [101] enforces that the kernel loads files whose hashes are certified with digital signatures stored in the file system (Figure 2.4). The application execution is halted until a dynamic library is loaded, and fails if the library fails the integrity check. IMA enables auditing by maintaining an *IMA log*, a dedicated file storing hashes of all files loaded to the memory since the kernel load. It adds each file to the IMA log and stores a hash over it in the TPM before the file is loaded to the memory. Any tampering of the IMA log is detectable because the IMA log's integrity hash must match the value stored in the TPM.

Enable remote attestation to prove that secure boot and integrity enforcement are enabled.

The TPM remote attestation protocol [91] delivers a technical assurance of the computer's integrity. The TPM chip digitally signs a report (*quote*) certifying hashes recorded since the computer boot. The hashes reflect loaded firmware and kernel and prove that integrity enforcement mechanisms are enabled. The verifier can check that the quote has not been manipulated because the TPM signs the quote with a signing key that is embedded in the TPM and linked to the certificate authority (CA) of the TPM manufacturer. However, the monitoring system cannot merely rely on the TPM attestation protocol because the protocol is vulnerable to the cuckoo attack [205]. It is indistinguishable whether an untrusted operating system proves its integrity presenting a quote from a local TPM or impersonates a trustworthy operating system presenting a quote from a remote TPM.

Detect the cuckoo attack by authenticating the TPM with a secret random number.

The monitoring system must ensure that the quote originated from the local TPM, *i.e.*, the TPM that collected integrity measurements from the software components that booted the operating system on the underlying computer. We propose to extend the agent with the functionality of checking that it communicates with the local TPM. The general idea consists of sharing a randomly generated *secret* φ with the local TPM to identify it uniquely and then use the secret to authenticate the TPM (Figure 3.4). The main challenge is how to generate a secret and share it with the local TPM without revealing it to the adversary. Otherwise, the adversary can mount the cuckoo attack by sharing it with a remote TPM.

Protect the secret in the TPM by relying on the one-way cryptographic hash function.

The TPM contains dedicated memory registers, called platform configuration registers (PCRs), that have important properties; they cannot be written directly, but they can only be extended with a new value using a cryptographic one-way hash function. The operation can be expressed as: *PCR_extend(n,value): pcr[n] = hash(pcr[n]||value)*. We propose to extend the secret φ on top of the existing measurements stored in the PCR to achieve the following properties: (i) an adversary cannot extract the secret from the PCR value after the secret is extended to the PCR because the hash function result is not invertible; (ii) an adversary cannot reproduce the PCR value in another TPM without knowing the secret, or finding a collision in the hash function; (iii) after extending the TPM with the secret, the secret is no longer needed to identify the TPM because the PCR value extended with the secret is unique.

Leverage DRTM technology to provide a trusted and measured environment to access the local TPM.

We must ensure that the secret is shared with the local TPM securely. We do it in a trusted environment established by hardware technologies available in modern CPUs because these technologies also permit verification of the established execution environment's load-time integrity. Therefore, they allow detecting (post-factum) any secret extraction attempt, including software side-channel attacks, because such attacks require violating the kernel or initramfs load-time integrity.

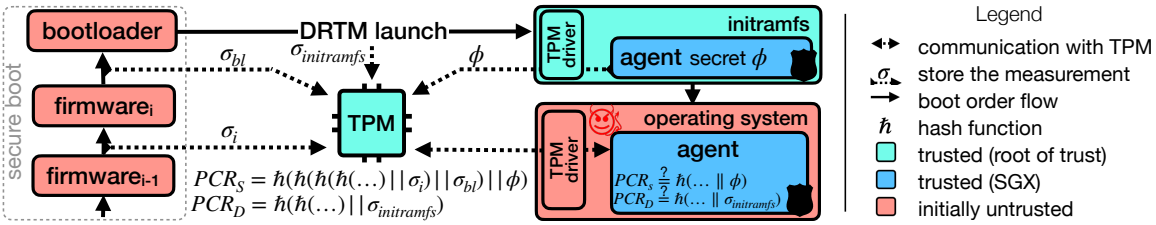


Figure 3.4. Defense against the cuckoo attack. The agent shares with the TPM a randomly generated secret φ, which is used later to authenticate the TPM. Platform configuration register (PCR) is TPM tamper-resistant memory.

We propose generating the secret and extending it to PCRs inside the initramfs[8] because DRTM allows for later verification of the kernel and initramfs integrity. Specifically, the DRTM [235], which is a hardware technology that establishes an isolated execution environment to run code on a potentially untrusted computer, can be used during the boot process (*i.e.*, by tboot [127]) to provide a measured load of the Linux kernel and initramfs.

The integrity measurements performed by DRTM cannot be forged because the TPM offers a dedicated range of PCRs (*dynamic PCRs*) that can only be reset or extended when the TPM is in a certain *locality* [132]; Only the code executed by DRTM can enter such locality. Therefore, the presence of measurements in dynamic PCRs confirms that the DRTM was executed, and the comparison of PCRs with the golden values confirms that the secret was shared with the local TPM because the correct TPM driver was used.

Leverage Intel SGX to transfer the golden TPM PCR value to the operating system runtime securely.

Once the secret is shared with the TPM, we must expose the unique local TPM's identifier (PCR value extended with the secret) to the agent running in the operating system. To do so, we leverage Intel software guard extensions (SGX) [45], a hardware CPU extension that provides confidentiality and integrity guarantees to the code executed in so-called *enclaves* in the presence of an adversary with root access to the computer. It offers a *sealing* [9] property that permits storing a secret on an untrusted disk where only the same enclave running on the same CPU can read it. The sealing and its revert operation *unsealing* use a CPU- and an enclave-specific key to encrypt and sign data in untrusted storage. We propose to communicate with the TPM from the inside of an enclave. First, the enclave executes in the initramfs where it shares a secret with the local TPM and seals the expected value of the TPM PCR to the disk. Then, it executes in the untrusted operating system, where it authenticates the TPM using the PCR value unsealed from the disk.

Leverage the SGX local and remote attestation to expose integrity measurements to the verifiers.

SGX offers local and remote attestation protocols [133]. While both protocols allow verifying that the expected code runs on a genuine Intel CPU, the SGX local attestation also permits

[8]The initramfs is a minimalistic root filesystem that provides a user space to perform initialization tasks, like loading device drivers, mounting network file systems, or decrypting a filesystem [207], before the operating system is loaded.

two enclaves to learn that they execute on the same CPU. We rely on this property to permit high-assurance security systems to establish trust with the agent running on the same computer. Like this, high-assurance security systems gain access to integrity measurements of the surrounding operating system. Similarly, central monitoring services leverage the SGX remote attestation to establish trust with agents.

3.5 CHORS architecture

3.5.1 High-level Overview

Figure 3.5 shows a high-level overview of the CHORS architecture, which consists of five entities. A security officer (❶) uses a controller (❷) to define security policies describing correct (trusted) operating system configurations. The controller communicates with agents (❸) running on every computer to check whether high-assurance security systems (❹) are executed in a trusted environment defined in security policies. Both the controller (❷) and the high-assurance security system executing inside SGX (❹) systematically query the agent to check if the operating system's integrity conforms to the criteria defined inside a security policy. Note that the integrity measurements are not aggregated or verified centrally. Instead, agents aggregate them and verify them locally on computers. Agents verify their location using trusted beacons (❺), services running in a known geographical location, *i.e.*, specific DCs.

We distinguish between two types of verifiers communicating with agents: local and remote verifiers. A local verifier is a high-assurance security system that requires strong confidentiality guarantees (❹). An example of such a service is a key management system [37, 159, 88] that executes inside an SGX enclave to protect integrity and confidentiality against privileged adversaries. The local verifier detects violations of the operating system's integrity by communicating with the agent running on the same host.

A remote verifier, *e.g.*, (❷), is an application running on a different computer than the agent. It aims to verify that the remote computer is located in the specific DC and its operating system is in the expected state. Typically, a remote verifier checks the integrity of the distributed system's deployment, *i.e.*, various services distributed over machines, data centers, and availability zones. The controller has broader knowledge about the network load, machine failures, service migrations, software updates. It helps the security officer to manage

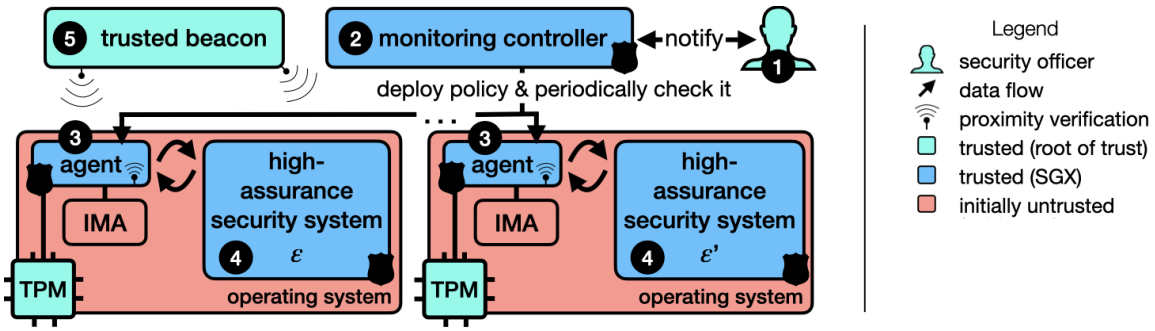


Figure 3.5. CHORS high-level architecture. The agent provides integrity measurements certified by the local TPM. The agent detects the cuckoo attack. High-assurance security system ε and the monitoring controller query the agent to ensure the integrity enforcement is enabled, thus, prevent software side-channel attacks. Agents use the trusted beacon to verify their geolocation.

the deployment while relying on individual services to react autonomously to integrity violations. The controller might be part of the security information and event management (SIEM) system that correlates system behavior to detect multi-faceted attacks [24].

Listing 3.1: Example of the CHORS's security policy

```
chain: |-
  -----BEGIN CERTIFICATE-----
  # TPM manufacturer certificates
  -----END CERTIFICATE-----
whitelist:
  - pcrs:
# secure boot / measured boot, PCRs 0-9
      - {id: 0, sha256: ff0c...e3}
      - {id: 3, sha256: e850...3e}
# trusted boot (DRTM) PCRs, 17-19
      - {id: 18, sha256: f9d0...cb}
      - {id: 19, sha256: a1e7...00}
runtime:
  certificate: |-
      -----BEGIN CERTIFICATE-----
      # IMA uses this certificate to verify signatures
      -----END CERTIFICATE-----
  software:
      - name: agent-0.8.0
        whitelist:
          840f...72: /bin/agent
      - name: AppArmour
        whitelist:
# hash of the executable
          1e73...f6: /sbin/apparmour
# hash of the configuration file
          c39e...34:  /etc/apparmour
location:
  - host: https://datacenter:10000/beacon
    max_latency: 10 # in milliseconds
    chain: |-
      -----BEGIN CERTIFICATE-----
      # TLS certificate chain of the trusted beacon
      -----END CERTIFICATE-----
```

3.5.2 Policy

The security officer defines security policies (*e.g.*, Listing 3.1) to declaratively state what software and dynamic libraries are permitted to run on the computer and what is the proper operating system configuration. He creates distinct security policies for each high-assurance security system. For example, a key management system has a different policy than a system processing medical data because they use different dynamic libraries, software, and operat-

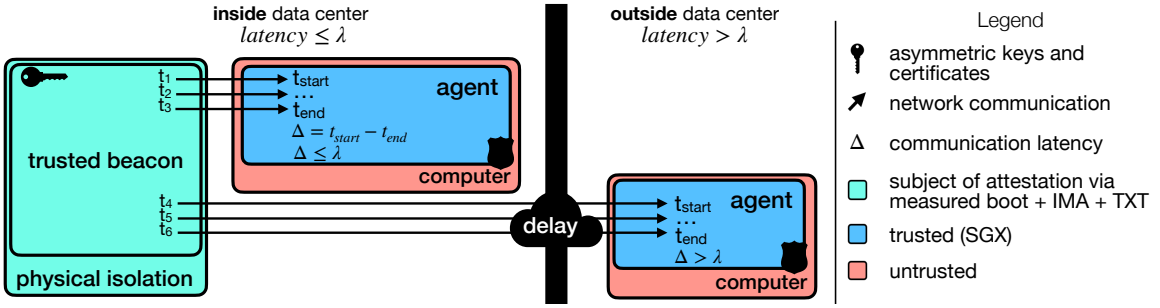


Figure 3.6. Trusted beacons. Agents rely on the trusted beacon to check that they are located in the expected data center. Only machines located inside the same data centers can achieve very low network latency required to prove their proximity.

ing system configurations. The monitoring controller reduces the burden of creating policies by allowing defining templates that can be combined to build individual policies with overlapping configurations. For example, services running on the same type of operating system share the same template that describes software and configuration specific to that operating system.

The agent uses the security policy to verify the operating system's integrity. The operating system is trusted if and only if the load-time integrity measurements of the kernel and the load-time integrity measurements of files loaded to the memory during the operating system runtime are declared on the whitelist or their corresponding digital signatures are verifiable using the certificate declared in the policy.

In more detail, the agent uses the TPM manufacturer's CA certificate chain to verify that the TPM chip attached to the computer is legitimate (line 1). The integrity of firmware and its configuration is represented as a whitelist of static PCRs (lines 8-9), while the integrity of the Linux kernel and the initramfs is specified as a whitelist of dynamic PCRs (lines 11-12). Trusted configuration files, executables, and dynamic libraries are defined in the form of hashes (lines 18-27) and a signing certificate (line 14). Software updates are supported via complementary solutions [204, 23] and require specification of additional certificate in the policy (line 14).

3.5.3 Trusted Beacon

A policy might constrain the computers' proximity to the well-known trusted beacons deployed in DCs (lines 29-34). A trusted beacon is a network service that responds to agents' requests with the current timestamp. The agent can then estimate the physical machine's proximity by measuring the network communication's round-trip times. The adversary cannot accelerate network packets enough to achieve a very short round-trip time achievable only between machines in the same local network.

Figure 3.6 shows a high-level view of the trusted beacon proximity verification protocol. The trusted beacon contains the asymmetric keypair with a certificate issued by a trusted authority, *e.g.*, a DC owner. These credentials, known only to the trusted beacon, prove that the DC owner placed the trusted beacon in the DC, and the trusted beacon executes in a trusted environment. The agent establishes trust with the trusted beacon by reading timestamps signed by the trusted beacon. The agent then estimates the network latency by calculating a trimmed mean from the differences between timestamps obtained from pairs of

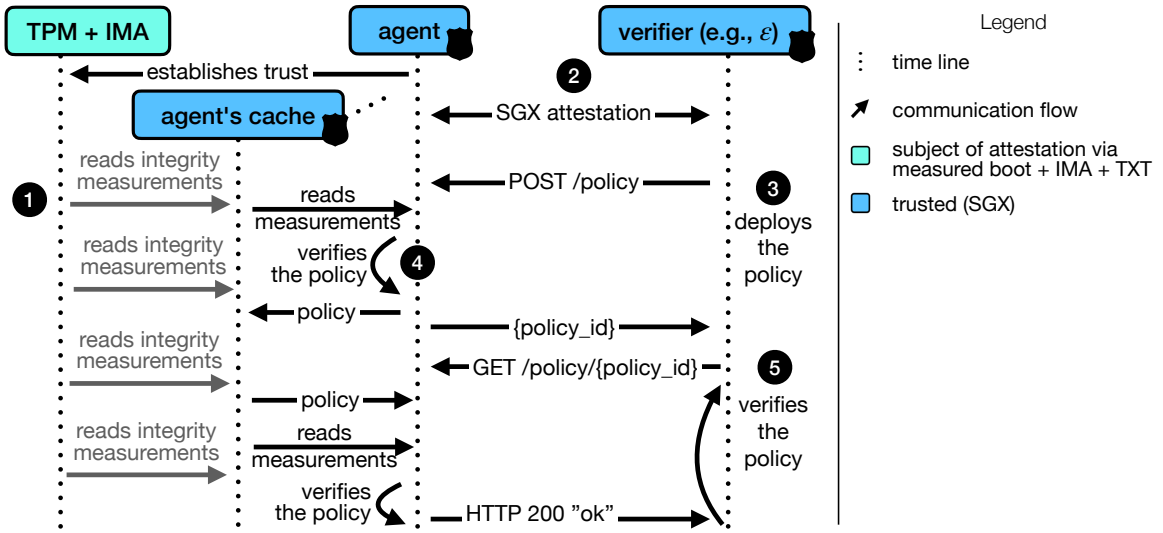


Figure 3.7. CHORS policy verification protocol. The agent maintains a separate thread (agent's cache) to constantly read the platform's fresh integrity measurements. Verifiers query the agent in parallel to ensure the compliance of the platform to the policy.

consecutive requests. A trimmed mean allows for tolerating network latency fluctuations because it excludes outliers.

Our design does not restrict what security mechanisms must protect the trusted beacon. In particular, the trusted beacon could be a network-accessible hardware security module (HSM) [109] returning signed timestamps. HSM is a crypto coprocessor offering the highest level of security against software and hardware attacks. It is embedded in a tamper responsive enclosure to actively detect physical and hardware attacks and protect against side-channel attacks. A cheaper but less secure alternative might run a TEE-based application implementing the abovementioned protocol over TLS. Related work [53] demonstrated that the network communication round-trip time between two SGX enclaves located in the same network take in average 264 μs, a latency not achievable from the outside of the data center.

3.5.4 Policy Verification Protocol

We designed the agent to act as a facade between the verifier and the TPM to enable multiple verifiers to check the operating system's integrity concurrently. Figure 3.7 shows how a verifier uses the policy verification protocol to attest to the operating system's integrity. The agent regularly reads the list of new software loaded by the operating system, the quote, and persists it into the *cache* that reduces the policy verification latency for future requests (❶). The local or remote verifier perform the SGX local or remote attestation [133] to verify the agent's identity and integrity and the CPU genuineness. The local attestation also proves that the agent runs on the same CPU (❷). Once the verifier deploys the policy (❸), the agent checks that the computer complies with the policy, stores the policy, and returns the corresponding *policy_id* (❹). The verifier uses the *policy_id* to re-evaluate the policy during future health checks (❺).

3.6 Implementation

We implemented CHORS on top of the Linux kernel. We use existing integrity enforcement mechanisms built in the Linux kernel, *i.e.*, IMA-appraisal, kernel module signature verification, and AppArmor. We rely on the support for the secure boot built-in the underlying firmware. We developed remote attestation components, *i.e.*, the agent in memory-safe language Rust [180]. We implemented the cuckoo attack detection mechanism and the policy verification protocol inside the agent. The monitoring controller allows defining policies, verifying the remote computer system's integrity, and alerting about integrity violations. We rely on the SCONE framework [11] and the SCONE cross-compiler to run CHORS inside the SGX enclave.

3.6.1 Computer Bootstrap

Figure 3.8 illustrates the bootstrap of a computer where the agent collects information required to detect the cuckoo attack. Consecutive unified extensible firmware interface (UEFI) components execute in the chain of trust; their integrity measurements are extended in static PCRs (❶). UEFI loads the bootloader, which starts the tboot (❷). The tboot leverages Intel trusted execution technology (TXT) [87, 44]–which implements DRTM on Intel CPUs–to establish a trusted environment. The tboot measures the integrity of the Linux kernel and initramfs, extends these measurements to dynamic PCRs (❸), and executes them (❹).

The initramfs has two essential properties; its integrity is reflected in dynamic PCRs, and failures during initramfs execution prevent machine booting. We rely on these properties to verify that the agent completed its execution. We refer to the agent execution inside initramfs as agent initialization (❺).

During the agent initialization, the agent requests the TPM to create a new attestation key (AIK), return the TPM's endorsement key (EK) certificate, and return the quote certifying PCRs (❻). The agent performs the *activation of credential* procedure ([12] p. 109-111) to verify that the AIK was created by the TPM, which possesses the private key associated with the EK certificate. The agent then *obfuscates* static PCRs by extending them with a random number generated inside the SGX enclave (❼). To ensure that the obfuscation succeeded and the boot process to continue, the agent reads PCRs again and compares them to the expected pre-computed hashes. After all, the AIK, the EK certificate, the TPM clock (includes computer reboot counter), and PCRs (original and obfuscated) are persisted in the file system in the SGX sealed *configuration file* (❽). The initramfs handles control to the operating system (❾), after the agent initialization finishes. The operating system executes the agent together with startup services. We refer to the agent execution after the operating system executes as agent runtime.

3.6.2 Establishing Trust

During the agent runtime, the agent verifies that there was no cuckoo attack during agent initialization and agent runtime by ensuring that the following conditions are fulfilled:

Condition 1: the agent is able to unseal the configuration file (❿). Relying on the properties of the SGX unseal, we conclude that the configuration file was created by the agent enclave running the same binary, and both enclaves were executed on the same SGX processor.

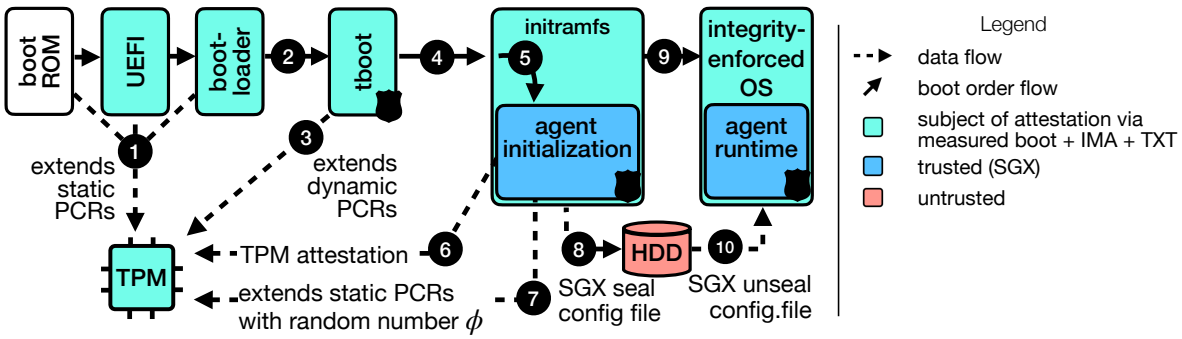


Figure 3.8. The platform boot process. To make the cuckoo attack detectable, the agent executes twice. First, in agent initialization, the agent executes in the measured environment where it shares a secret with the TPM. Second, in agent runtime, the agent establishes trust with the local TPM or detects the cuckoo attack.

Condition 2: a successful match between dynamic PCRs read from the TPM and the golden dynamic PCRs. It proves that during agent initialization, the agent enclave was executed in the trusted environment (Linux kernel, initramfs, and correct TPM driver), and it successfully obfuscated the TPM.

Condition 3: a successful match of static PCRs read from the TPM with obfuscated static PCRs read from the configuration file. It proves that the configuration file contains the information gathered earlier from the same TPM.

Condition 4: a successful match of the reboots counter stored in the configuration with the reboots counter value read from the fresh quote proves that the computer did not reboot since the agent initialization.

Finally, considering conditions 1, 2, 3, 4, and what they indicate once fulfilled, we conclude that the quote was issued by the TPM that collected software measurements during the computer bootstrap. [203] formally proves this claim.

3.6.3 Cache Updates

To decrease the policy verification latency, the agent starts a separate thread reading the computer state to validate it against future policy verification requests. The agent recurrently retrieves the quote and verifies that the quote certifies PCRs values read during the agent initialization, and it repeatedly reads new events from the IMA log.

Hashes of all events are stored in the enclave's memory, together with the number of bytes read (B), and the last value of IMA PCR (D). To read new events, the agent first retrieves the quote and opens the IMA log file skipping B bytes. It then reads a new event from the file and recalculates the integrity hash by extending D with the event's hash. This process is repeated for each new event and finishes when the integrity hash is equal to the hash of the IMA PCR retrieved from the quote. If the agent reaches the end of the IMA log and the integrity hash does not match the hash in the IMA PCR, it detects the tampering of the IMA log and the operating system is considered compromised.

3.6.4 Policy Verification

The agent exposes the policy verification functionality via a TLS-protected representational state transfer (REST) application programming interface (API) endpoint to simplify the com-

munication interface between verifiers and agents. It is enough for verifiers to check the agent's identity by verifying its X.509 certificate presented during a TLS-handshake. Currently, TLS credentials are delivered to the agent via a key management system (KMS) [88]. As future work, the agent will create a self-signed certificate via sgx-ra-tls [149], thus excluding the KMS from the trusted computing base. The verifier can also rely on the SGX remote attestation [133] to ensure the agent's identity and integrity.

The agent stores a once deployed policy in the in-memory key-value map under a randomly generated key *policy_id* to permit tenants to verify the same policy again. The agent can be queried with the policy_id to verify that the operating system integrity has not changed since the last verification. An adversary cannot change once deployed policy because SGX protects the agent's memory from tampering, *i.e.*, SGX guarantees integrity, confidentiality, and freshness of data.

3.7 Security Risk Assessment

CHORS combines different security techniques to build a framework providing technical assurance that applications execute inside TEE on the trustworthy operating system. However, each technique operates under a different threat model, and a careful analysis of existing attacks is required to claim security guarantees.

3.7.1 Preventing Physical and Hardware Attacks

First of all, applied techniques usually do not protect against hardware and physical attacks. The TPM is vulnerable to simple hardware attacks on the communication bus with the CPU that allows an adversary to reset the TPM [142], reply to arbitrary measurements [238], including measurements corresponding to the DRTM launch [272]. Similarly, Intel SGX is vulnerable to clock speed and voltage manipulation [189]. Direct memory access attacks [178] or cold-boot attacks [100] can compromise the entire operating system and applications that store data in the main memory in plaintext. To prevent these kinds of attacks, unlike other works [283, 95], we propose to attest to the physical location of the computer. Regulators require that DCs are access controlled and place computers inside security cages [77]. We argue that these techniques provide enough security to consider physical and hardware attacks inside the trusted data center negligible.

We use the concept of a trusted beacon to verify that the computer is located in the trusted DC. In the real-world, the trusted beacon functionality could be provided by a hardware security module [109] or a trusted timestamping authority running on a computer with formally proved software [148, 212]. The only assumption is that trusted beacons must be securely placed inside the DC and then be protected from being moved.

3.7.2 Establishing Trust with the Agent

To verify that the computer is indeed located in the expected DC, we must rely on the agent executing on a potentially untrusted computer exposed to physical and hardware attacks. To authenticate the agent and verify that it executes on a genuine Intel SGX CPU, we leverage Intel SGX remote attestation [133]. In the past, researchers managed to extract Intel SGX attestation keys [262, 260] that allowed impersonating a genuine SGX CPU. The available mitigations are: i) relying on on-premise data center attestation mechanism [228], ii) checking

for revoked SGX attestation keys, and iii) verifying that the agent runs in the proximity of a trusted device to ensure that it is in the correct data center composed of legitimate SGX machines [53]. In all cases, we must trust the CPU manufacturer, SGX design, cryptographic primitives, and CPU implementation. We consider these assumptions practical because they are common industry practices.

3.7.3 Establishing Trust with the TPM

CHORS relies on TXT, SGX, and TPM to detect the cuckoo attack. Researchers demonstrated that malware placed in the system management mode (SMM) could survive the TXT late launch [273]. To mitigate attacks on SMM, Intel introduced an SMI transfer monitor that constrains the system management interrupt handler mitigating this class of attacks entirely [191]. Other TXT-related and tboot vulnerabilities [274, 103] were related to memory vulnerabilities in Intel's firmware and tboot implementations. These vulnerabilities have been patched as part of a software update release cycle.

Intel SGX is vulnerable to microarchitectural and side-channel attacks that violate SGX confidentiality guarantees [260]. Some of these attacks led to the leakage of Intel attestation keys which might be used to forge the SGX attestation [260]. Intel constantly patches the vulnerabilities with microcode updates or hardware changes. The presence of microcode updates is reflected in the SGX attestation quote, allowing the verifier to check that the enclave executes on the patched processor. Intel also invalidates attestation keys that might have been leaked, preventing usage of these keys for attestation. Nonetheless, we do consider these attacks as a real threat because of their severity and the multitude of variants that appear.

These attacks do not impact CHORS guarantees because they only affect SGX confidentiality and not integrity, assuming leaked attestation keys are properly revoked by Intel or on-premise data center attestation mechanisms [260, 228]. The only security-sensitive data that might be used to compromise CHORS is the secret shared between the agent and the TPM. However, the secret lives only during the agent initialization, where the presence of malware is detected. In more detail, an adversary can extract the secret shared between the agent and the TPM during the agent initialization to mount the cuckoo attack by sharing the secret with an arbitrary TPM. We formally proved [203] that the CHORS protocol is immune to these kinds of attacks because the agent detects that the secret was leaked once it executes in the *agent runtime*. The agent detects that malware was present during the agent initialization because both initramfs and kernel are measured by DRTM, and their measurements are securely transferred to the agent in the *agent runtime* via SGX sealing. An adversary cannot tamper with the sealed data because only the same enclave running on the same CPU can seal and unseal the data. Thus, the presence of malware and secret leakage are revealed.

3.7.4 Establishing Trust with the Operating System

Because the agent can read the load time integrity of the kernel stored inside the dynamic PCR in the TPM, it can ensure that the computer executes a kernel that was intended to load because even if an adversary boots a malicious kernel, she cannot tamper with PCRs that reflect the malicious kernel load.

An adversary who gains access to the computer by stealing credentials using social engineering or exploiting a misconfiguration cannot run arbitrary software because she does not

have the signing key to issue a certificate required by the integrity-enforcement mechanisms (IMA) to authorize the file.

However, an adversary might exploit memory vulnerabilities in the existing code, such as Linux kernel or software executing on the system remotely [35]. This is feasible because most system software is implemented in unsafe memory languages. We assume that the operating system owner relies on an additional security mechanism enumerated in §3.3 to enforce the runtime process integrity. Typically, the system owner also minimizes the trusted computing base (TCB) by authorizing only crucial software to run on a computer. He does it by digitally signing only trusted software and relying on the IMA-appraisal to enforce it during the operating system runtime.

An adversary who gains access to the computer can restart it and disable the security mechanisms or boot the computer into an untrusted state. In §3.8.2, we estimate the vulnerability window size in which the monitoring controller detects the computer integrity violation.

Another attack vectors are network side-channel attacks, such as NetCAT [164], and rowhammer attacks over the network [246]. In these attacks, an adversary does not have to run malware on the computer but instead sends malicious network packages that modern network cards place directly in the main memory. We assign a low risk to these classes of attacks because (i) they are hard to perform in a noisy production environment, (ii) they are detectable by network traffic monitoring tools and firewalls because they generate high network activity, (iii) mitigation techniques exist and can be applied independently [164, 246].

3.8 Evaluation

We evaluate CHORS in three-folds. In §3.8.1, we demonstrate CHORS protecting a real-world application from the eHealth domain. Then, in §3.8.2 and §3.8.3, we evaluate CHORS' security and performance, respectively. The evaluation of the real-world application applies only to §3.8.1.

Testbed. Experiments execute on a rack-based cluster of three Dell PowerEdge R330 servers connected via a 10 Gb Ethernet. Each server is equipped with an Intel Xeon E3-1270 v5 CPU, 64 GiB of RAM, Infineon 9665 TPM 2.0, running Ubuntu 16.04 LTS with Linux kernel v4.4.0-135-generic. The CPUs are on the microcode patch level (0xc6). The enclave page cache (EPC) is configured to reserve 128 MiB of RAM. During all experiments, the agent, the monitoring controller, and the trusted beacon run on different machines.

Table 3.1. The execution time of the eHealth application. Mean values calculated from 30 independent application executions. The standard deviation in all variants was 1 sec.

	native	SCONE	CHORS
Execution time	41 sec	52 sec	53 sec
Security level			
- tolerate rogue operator	✗	✓	✓
- tolerate untrusted OS	✗	✓	✓
- no side-channel attacks (exclusive access to the OS)	✗	✗	✓
- data processed in correct geolocation	✗	✗	✓

3.8.1 Protecting a Real-world eHealth Application

We leveraged CHORS to protect an eHealth application provided to us by a partner who requires protection of his intellectual property (the application's source code) and the confidentiality of the privacy-sensitive patients' data. This dataset contains concentrations of 112 metabolites in cerebrospinal fluid samples from patients with bacterial meningitis, viral meningitis/encephalitis, and non-inflamed controls. The application, implemented in Python, uses a machine learning (ML) algorithm to understand pathophysiological networks and mechanisms as well as to identify disease-specific pathways that could serve as targets for host-directed treatments to reduce end-organ damage. We used publicly available SCONE docker images [232] to run the application inside a container executed inside the SGX enclave. We configured the operating system to use IMA and run the CHORS's agent. On two other machines, we deployed the trusted beacon and the monitoring controller, which was constantly querying the agent to verify the operating system integrity.

We measured the execution time of the machine learning algorithm run in three different variants; in *native*, the application executes in the untrusted operating system; in *SCONE*, the application executes in the untrusted operating system but inside an SGX enclave provided by SCONE; in *CHORS*, the application executes inside an SGX enclave on an integrity-enforced operating system booted with CHORS.

Table 3.1 shows that the machine learning algorithm's execution inside the SGX enclave takes 52 sec, which was 1.3× longer than the native execution (41 sec). CHORS further increased the application execution time by 2%, compared to the SGX enclave execution. This is an acceptable performance overhead, assuming the higher security guarantees offered by CHORS and the compliance with the privacy regulations required by the EU law.

3.8.2 Security

An adversary cannot violate the computer system's integrity if all integrity enforcement mechanisms are properly configured and enabled (including mechanisms protecting runtime process integrity §3.3) because the kernel rejects untrusted files from loading to the memory. However, an adversary can run arbitrary software if she gets enough privileges to boot the computer with disabled enforcement mechanisms. We run a set of micro-benchmarks to estimate the *vulnerability window size* expressed with Equation (1), during which the integrity violation remains undetected.

$$t_{vw} = t_{rq} + 2 * (nt_{re} + t_{vp}) \tag{3.1}$$

t_{vw} is the vulnerability window size, t_{rq} is the time to read a TPM quote, n is the maximum number of events that can be opened within t_{rq}, t_{re} is the time to read a single event from the IMA log, t_{vp} is the time required by the agent to verify the policy and by a verifier to send, receive, and process the verification request.

What is the latency of reading a TPM quote?

Each time the agent reads the IMA log, it reads a fresh TPM quote to verify the IMA log's integrity. The TPM supports different signing schemes that have a direct impact on the TPM quote read latency.

Table 3.2. The latency of reading the TPM quote generated using different signing schemes. Mean values calculated from 30 experiment executions. σ stands for standard deviation.

Signing scheme	TPM quote read latency
RSA 2048 with SHA-256	521 ms ($\sigma = 4$ ms)
ECDSA P256 with SHA-256	155 ms ($\sigma = 2$ ms)
HMAC with SHA-256	107 ms ($\sigma = 3$ ms)

Table 3.2 shows that TPM issues a quote using hash-based message authentication code (HMAC) in 107 ms, which is 4.9× faster than when using Rivest-Shamir-Adleman (RSA) cryptography and 1.4× faster when using elliptic curve digital signature algorithm (ECDSA). Thus, selecting an HMAC or ECDSA allows validating the IMA log's integrity faster than when using RSA. We assume usage of the ECDSA when reading a quote, thus t_{rq}=155 ms.

Table 3.3. The latency of reading a single event from the IMA log. Mean values calculated from 1200 events readings. σ stands for standard deviation.

	Read latency of a single IMA log entry
ImaNg event	34 μs ($\sigma = 28\,\mu$s)
ImaSig event	58 μs ($\sigma = 32\,\mu$s)

What is the latency of reading integrity measurements?

We measured the latency of reading new measurements from the IMA log to learn how fast the agent can detect the integrity violation. During the first read of the IMA log, the agent reads all measurements collected by IMA during the operating system boot, which is typically the biggest chunk of the IMA log that has to be read by the agent at once. The bootstrap of Ubuntu Linux produces approximately 1800 measurements. The agent needs 130 ms to read all events from the IMA log, recalculate the IMA log integrity hash, and compare the hash to the PCR value.

After the initial IMA log read, the agent reads only the new IMA measurements since the last IMA log read. The time needed to read the integrity measurements depends on the number of new events measured and added to the IMA log.

Table 3.3 shows that the agent requires 34 μs and 58 μs to retrieve a single ImaNg and ImaSig event, respectively. The ImaNg, a default IMA event format providing the file's integrity hash. The ImaSig event entry extends the ImaNg format by also including the file's signature. So, the maximum event read time t_{re}=58 μs.

How much time does it take to detect the integrity violation?

The vulnerability window for the attack consists of the time the agent takes to read a fresh quote, retrieve new events from the IMA log, and process the policy verification request. We assume that when the agent reads a quote (t_{rq}), an adversary can cause IMA to open no more than n=3875 files (according to our measures, opening a file takes at least 40 μs). The agent would require about $n * t_{re}$=225 ms to read events, and about t_{vp}=100 ms to verify

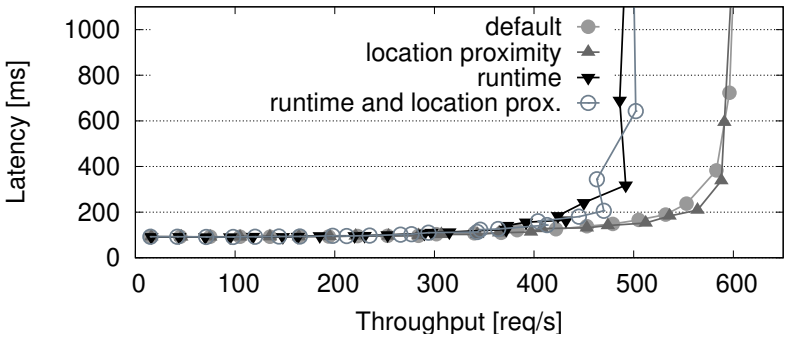


Figure 3.9. Policy verification throughput. *Default* policy checks secure boot and trusted boot. *Location proximity* checks geolocation. *Runtime* verifies IMA measurements.

them against the policy, see §3.8.3. Therefore, using Equation (1), we estimate that the policy verification protocol has a vulnerability window of approximately t_{vw}=805 ms.

3.8.3 Performance

How scalable is CHORS? Can it efficiently verify policies on behalf of multiple verifiers?

In our design, the agent is the security-critical component that performs local integrity attestation on behalf of high-assurance security systems, centralized monitoring services, and security officers. To verify the agent's ability to verify security policies, we measured the policy verification throughput – the time in which the agent responds to the verifier's request verifying operating system integrity. Our experiments compare four variants of the policy content: (i) *default*, the policy contains only the definition of static and dynamic PCRs; (ii) *location proximity*, the default policy content with additional constraints about proximity to trusted beacon; (iii) *runtime*, the default policy content with a whitelist of trusted software; (iv) *runtime and location proximity*, the combination of the runtime and location proximity policies.

Figure 3.9 shows that the agent achieves the maximum throughput of 623 req/sec when verifying a default policy. A similar throughput is achieved for the policy with the location proximity extension. The throughput decreases to 521 req/sec when the agent verifies a security policy containing IMA measurements because of the overhead caused by reading new IMA measurements. An optimal latency of 100 ms is achieved for all policy variants when the throughput < 250 req/sec.

Table 3.4. The mean remote attestation latency comparison between different integrity monitoring frameworks. In all systems, the TPM quote was signed with RSA signing scheme. *se* stands for standard error.

	Remote attestation latency
CHORS	665 ms (se=2 ms)
Intel CIT	2475 ms (se=5 ms)
IBM ACS	5677 ms (se=22 ms)

How does CHORS performance compare to the existing monitoring frameworks?

We measured the integrity verification latency of the existing integrity monitoring frameworks to check if the presented framework can be considered practical in terms of performance. Specifically, we compared CHORS with Intel open cloud integrity technology (Intel CIT) [128, 125], and IBM TPM attestation client-server (IBM ACS) [111], which is a sample code for a Trusted Computing Group (TCG) attestation application. We measured the total time taken to establish a connection with an agent, retrieve a fresh quote, and compare PCRs with a whitelist. In all experiments, the TPM has been previously commissioned.

Table 3.4 shows that CHORS with the mean latency of 665 ms outperforms Intel CIT by 3.7× and IBM ACS by 8.5×. CHORS achieves better performance because, during the initialization, it caches AIK, static PCRs, and dynamic PCRs that do not change during the entire agent's life cycle. The agent verifies that those values did not change by comparing them to the certified values obtained from the quote. Furthermore, unlike others, the agent verifies the integrity of the IMA log and PCRs by recomputing a hash over cached PCRs and IMA log and matching it against the PCRs hash in the quote. It allows the agent to skip the slow process of reading PCRs and, consequently, reduce communication with the TPM to a single recurrent quote read operation.

Table 3.5. The latency of the policy deployment into the agent depending on the content of the security policy. Mean values calculated from 600 independent policy deployments. σ stands for standard deviation.

Security policy content	Deployment latency
Static and dynamic PCRs	576 ms ($\sigma = 15$ ms)
+ location proximity	626 ms ($\sigma = 17$ ms)
+ IMA measurements	606 ms ($\sigma = 16$ ms)
+ location prox. and IMA measur.	677 ms ($\sigma = 15$ ms)

How much time does it take to deploy a single security policy?

Table 3.5 shows the latency of the policy deployment protocol using different policy extensions. The latency is measured as the total time between establishing a transport layer security (TLS) connection with CHORS, a policy upload, a verification using a fresh quote, and a response retrieval. The default policy's size, containing the whitelist of 13 PCRs and one TPM manufacturer's CA certificate, is 4.7 kB. Its deployment takes 576 ms. The runtime policy size, containing the whitelist of 1790 files and an IMA signing certificate, is 235 kB (50× the default policy). Its deployment lasts 606 ms, which is only a 1.05× of the default policy deployment latency. The deployment latency of a policy with the location proximity extension depends on the communication latency between CHORS and trusted beacons. The deployment of the policy with one trusted beacon located in the same data center takes 626 ms.

How does CHORS impact the boot time of a computer?

We used the *systemd-analyze* tool to measure the load time of initramfs and userspace in different configuration variants of Ubuntu. Figure 3.10 shows that the native Ubuntu Linux starts in 19 sec, from which the load of the userspace takes 13 sec and the kernel with

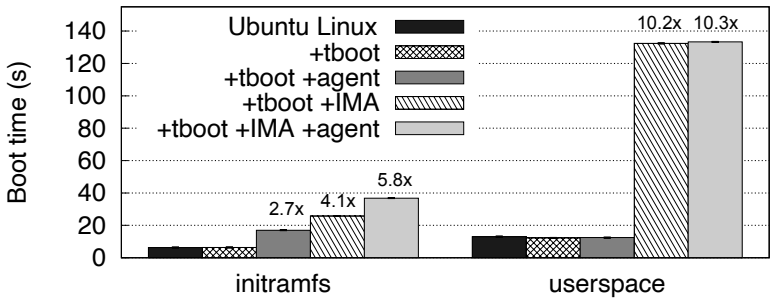


Figure 3.10. Impact of CHORS on boot time.

initramfs remaining 6 sec. tboot executes after the bootloader and before the initramfs, thus not influencing the load time of the operating system. The activation of IMA configured to measure all files defined by the TCG group (ima_tcb boot option), increases the boot time to 158 sec, 8.3× of the native. A load of userspace takes 84% of this time, which is caused by the measurement of 1790 files. The boot time could be decreased by reducing the number of services loaded by the operating system. CHORS increases the boot time by 58% compared to the Ubuntu Linux with tboot and 8% compared to the Ubuntu Linux with IMA. The increased boot time is mostly caused by the execution of time-consuming TPM operations in initramfs performed by CHORS and IMA.

3.9 Related Work

Like the existing monitoring systems [128, 111], CHORS relies on the TPM attestation protocol to verify the computer's integrity. Unlike them, CHORS is resilient to the cuckoo attack. Existing defenses against this attack have a limited application for high-assurance security systems. Fink et al. proposed a time side-channel approach [67] to detect the cuckoo attack. As confirmed by the authors, it is prone to false positives and requires stable measurement conditions, an impractical assumption in real-world scenarios. Flicker [184] accesses local TPM from the isolated execution environment established by DRTM. However, DRTM does not attest to the computer location which makes its attestation untrustworthy due to simple hardware attacks [272]. Moreover, DRTM permits executing only a single process on the entire CPU at the same time. This impacts application's throughput because a single context switch to DRTM-established environment takes 10-100s of milliseconds [183]. CHORS instead first verifies that the computer is in the trusted data center (thus, no hardware attacks are possible) and uses DRTM only once when provisioning the TPM. This approach provides better performance as required by modern applications.

Other solutions for root of trust identification problem require the verifier to solve biometric challenge [52], observing emitted LED signals [244], verifying the device state displayed on the screen [51, 165], using trusted devices to scan bar codes sealed on the device [182], or pressing a special-purpose button for bootstrapping trust during the computer boot [205]. These approaches have limitations because (i) the TPM is a passive device controlled by software which, due to lack of trusted I/O paths to external devices, can redirect, reply, or fool the communication, and (ii) they require human interaction and thus do not scale for the

DC-level.

Recently, Dhar et al. proposed ProximiTEE [53] to deal with the SGX (*not* TPM) cuckoo attack by attaching a trusted device to the computer and detecting the cuckoo attack during the SGX attestation. This solution can verify that the SGX enclave executes on the computer with the attached trusted device because of the very low communication latency between the enclave and the device. Although, as denoted by Parno [205] this approach cannot be used to detect the TPM cuckoo attack because of the slow speed of the TPM, CHORS could use ProximiTEE as a trusted beacon implementation to prove that the computer is located in the expected data center.

Other work focuses on tolerating malware in the operating system while preventing side-channel attacks on TEEs. There are three approaches to mitigate these attacks: (i) static vulnerability detection [96, 201], (ii) attack prevention [4, 26, 75], and (iii) attack detection [200, 41]. The first one consists of analyzing and modifying source code to detect gadgets [96, 201]. However, finding all gadgets is difficult or impossible because the search narrows to gadgets specific to known attacks. The second approach prevents attacks by hiding access patterns using oblivious execution/access pattern obfuscation, resource isolation [75], or hardware changes [266]. These techniques address only specific attacks [75], require hardware changes [266], or incur large performance overhead [4, 26]. The last approach consists of runtime attack detection [200, 41] by isolating and monitoring resources of instrumented programs. But, it targets selected attacks and assumes some amount of statistical misses. CHORS aims at preventing such attacks without requiring source code changes or hardware modifications, with low performance overhead but a larger trusted computing base.

3.10 Summary

We responded to regulatory demands that require stronger isolation of high-assurance security systems by running them inside trusted execution environments on top of a trustworthy operating system and in the expected geolocation. We demonstrated that the combination of Intel SGX with TPM-based solutions meets such requirements but requires protection against the cuckoo attack. We proposed a novel deterministic defense mechanism against the cuckoo attack and formally proved it. We implemented a framework that monitors and enforces the integrity as well as geolocation of computers running high-assurance security systems and mitigates the cuckoo attack. Our evaluation and security risk assessment show that the CHORS is practical.

4 Remote Attestation of the Virtual Machine's Runtime Integrity

Chapter 3 introduced a technique allowing a computer owner to verify that his security-sensitive application executes on an integrity-enforced operating system running on a computer located in the data center under his control. However, many of today's systems run instead on computers owned and managed by another legal entity because it allows for cost reduction, *i.e.*, the responsibility of the computing resources maintenance and administration shifts from application owners (tenants) to the infrastructure owners (cloud providers). Trust is of paramount concern in such a setting, because software managing computing resources and its configuration and administration remains out of the tenant's control. Tenants have to trust that the cloud provider, its employees, and the infrastructure protect the tenant's intellectual property as well as the confidentiality and the integrity of the tenant's data. A malicious employee [211], or an adversary who gets into possession of employee credentials [113, 136], might leverage administrator privileges to read the confidential data by introspecting virtual machine (VM) memory [245] to tamper with computation by subverting the hypervisor [145], or to redirect the tenant to an arbitrary VM under her control by altering a network configuration [285]. We tackle the problem of how to establish trust in a VM executed in the cloud. Specifically, we focus on the integrity of legacy systems executed in a VM.

The existing attestation protocols focus on leveraging trusted hardware to report measurements of the execution environment. In trusted computing [73], the trusted platform module attestation [90] and integrity measurement architecture (IMA) [225] provide a means to enforce and monitor integrity of the software that has been executed since the platform bootstrap [89]. The virtual TPM (vTPM) [21] design extends this concept by introducing a software-based trusted platform module (TPM) that, together with the hardware TPM, provides integrity measurements of the entire software stack — from the firmware, the hypervisor, up to the VM. However, this technique cannot be applied to the cloud because an adversary can tamper with the communication between the vTPM and the VM. For example, by reconfiguring the network, she can mount a man-in-the-middle attack to perform a TPM reset attack [142], compromising the vTPM security guarantees.

A complementary technology to trusted computing, trusted execution environment (TEE) [81], uses hardware extensions to exclude the administrator and privileged software, *i.e.*, operating system, hypervisor, from the trusted computing base. The Intel software guard exten-

sions (SGX) [45] comes with an attestation protocol that permits remotely verifying the application's integrity and the genuineness of the underlying hardware. However, it is available only to applications executing inside an SGX enclave. Legacy applications executed inside an enclave suffer from performance limitations due to a small amount of protected memory [11]. The SGX adoption in the virtualized environment is further limited because the protected memory is shared among all tenants.

Alternative technologies isolating VMs from the untrusted hypervisor, *e.g.*, AMD secure encrypted virtualization (SEV) [140, 139] or IBM protected execution facility (PEF) [105], do not have memory limitations. They support running the entire operating system in isolation from the hypervisor while incurring minimal performance overhead [98]. However, their attestation protocol only provides information about the VM integrity *at the VM initialization time*. It is not sufficient because the loaded operating system might get compromised later–*at runtime*– with operating system vulnerabilities or misconfiguration [279]. Thus, to verify the runtime (post-initialization) integrity of the guest operating system, one would still need to rely on the vTPM design. But, as already mentioned, it is not enough in the cloud environment.

Importantly, security models of these hardware technologies isolating VM from the hypervisor assume threats caused by running tenants' operating systems in a shared execution environment, *i.e.*, attacks performed by rogue operators, compromised hypervisor, or malicious co-tenants. These technologies do not address the fact that a typical tenant's operating system is a complex mixture of software and configuration with a large vector attack. *I.e.*, the protected application is not, like in the SGX, a single process, but the kernel, userspace services, and applications, which might be compromised while running inside the TEE and thus exposes tenant's computation and data to threats. These technologies assume that tenants are responsible for protecting the operating system. However, they lack primitives to enable runtime integrity verification and enforcement of guest operating systems. This work proposes means to enable such primitives, which are neither provided by the technologies mentioned above nor by the existing cloud offerings.

4.1 Contribution

We overcome the limitations of the existing approaches by combining trusted computing techniques with TEE. We present TRIGLAV [9], a VM remote attestation protocol that provides integrity guarantees to legacy systems executed in the cloud. TRIGLAV has noteworthy advantages. First, it supports legacy systems with zero-code changes by running them inside VMs on the integrity-enforced execution environment. To do so, it leverages trusted computing to enforce and attest to the hypervisor's and VM's integrity. Second, TRIGLAV limits the system administrator activities in the host operating system using integrity-enforcement mechanisms while relying on the TEE to protect its own integrity from tampering. Third, it supports tenants connecting from machines not equipped with trusted hardware. Specifically, TRIGLAV integrates with the secure shell (SSH) protocol [280]. Login to the VM implicitly performs an attestation of the VM.

To summarize, in this chapter, we make the following contributions: (i) We demonstrated the security issues of applying trusted computing techniques in the cloud (§4.3.2). (ii) We showed how to mitigate these weaknesses by leveraging IMA, TEE, and key management (§4.4). (iii) We designed a protocol, TRIGLAV, attesting to the VM's runtime integrity (§4.4).

[9] In the Slavic mythology, Triglav (Trzygłów) is a powerful three-headed deity representing a fusion of three kingdoms: heaven, earth, and undergrounds [79].

(iv) We implemented the TRIGLAV prototype using state-of-the-art technologies commonly used in the cloud (§4.5). (v) We evaluated it on real-world applications (§4.6).

4.2 Threat Model

We require that the *cloud node* is built from the software which source code is certified by a trusted third party [198] or can be reviewed by tenants, *e.g.*, open-source software [8] or proprietary software accessible under a non-disclosure agreement. Specifically, such software is considered safe and trusted when (i) it originates from trusted places like the official Linux git repository; (ii) it passes security analysis like fuzzing [281]; (iii) it is implemented using memory safe languages, like Rust [180]; (iv) it has been formally proven, like seL4 [148] or EverCrypt [212]; (v) it was compiled with memory corruption mitigations, *e.g.*, position-independent executables with stack-smashing protection.

Our goal is to provide tenants with a *runtime integrity attestation protocol* that ensures that the cloud node (*i.e.*, host operating system, hypervisor) and the VM (guest operating system, tenant's legacy application) run only expected software in the expected configuration. We distinguish between an internal and an external adversary, both without capabilities of mounting physical and hardware attacks (*e.g.*, [272]). This is a reasonable assumption since cloud providers control and limit physical access to their data centers.

An *internal adversary*, such as a malicious administrator or an adversary who successfully extracted administrators credentials [113, 136], aims to tamper with the hypervisor configuration or with a VM deployment to compromise the integrity of the tenant's legacy application. She has remote administrative access to the host machine that allows her to configure, install, and execute software. The internal adversary controls the network. She can insert, alter, and drop network packages.

An *external adversary* resides outside the cloud. Her goal is to compromise the integrity of security-sensitive applications. She can exploit a guest operating system misconfiguration or use social engineering to connect to the tenant's VM remotely. Then, she runs dedicated software, *e.g.*, software debugger or custom kernel, to modify the legacy application's behavior.

We consider the TPM, the CPU, and their hardware features trusted. We rely on the soundness of cryptographic primitives used by software and hardware components. We treat software-based side-channel attacks (*e.g.*, [151]) as orthogonal to this work because of (i) the counter-measures existence (*e.g.*, [200]) whose presence is verifiable as part of the TRIGLAV protocol, (ii) the possibility of provisioning a dedicated (not shared) machine in the cloud (see §3).

4.3 Background and Problem Statement

4.3.1 Load-time Integrity Enforcement

A cloud node is a computer where multiple tenants run their VMs in parallel on top of the same computing resources. VMs are managed by a hypervisor, a privileged layer of software providing access to physical resources and isolating VMs from each other. Since the VM's security depends on the hypervisor, it is essential to ensure that the correct hypervisor controls the VM.

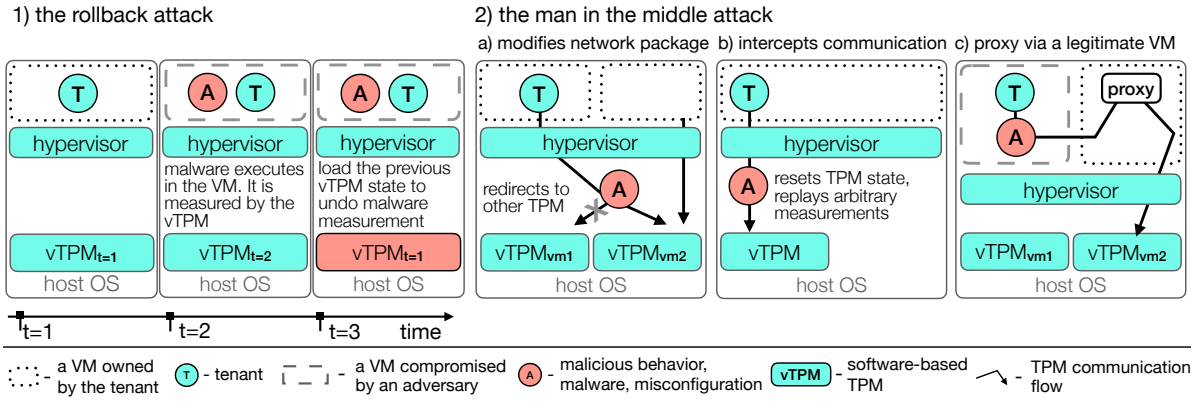


Figure 4.1. An adversary with root access to the hypervisor can violate the security guarantees promised by the vTPM [21] design.

4.3.2 Runtime Integrity Enforcement

The administrator has privileged access to the machine with complete control over the network configuration, with permissions to install, start, and stop applications. These privileges permit him to trick the dynamic root of trust for measurements (DRTM) attestation process because the hypervisor's integrity is measured *just once* when the hypervisor is loaded to the memory. The TPM report certifies this state until the next DRTM launch, *i.e.*, the next computer boot. Hence, after the hypervisor has been measured, an adversary can compromise it by installing an arbitrary hypervisor [222] or downgrading it to a vulnerable version without being detected.

Integrity measurement architecture (IMA) [89, 225, 101] allows for mitigation of the threat mentioned above. Being part of the measured kernel, IMA implements an *integrity-enforcement mechanism* [101], allowing for loading only digitally signed software and configuration. Consequently, signing only software required to manage VMs allows for limiting activities carried out by an administrator on the host machine. A load of a legitimate kernel with enabled IMA and input-output memory management unit (IOMMU) is ensured by DRTM, and it is attestable via the TPM attestation protocol.

4.3.3 Problems with Virtualized TPMs

The TPM chip cannot be effectively shared with many VMs due to a limited amount of platform configuration registers (PCRs). The vTPM [21] design addresses this problem by running multiple software-based TPMs exposed to VMs by the hypervisor. This design requires verifying the hypervisor's integrity before establishing trust with a software-based TPM. We argue that verifying the hypervisor's integrity alone is not enough because an administrator can break the software-based TPM security guarantees by mounting attacks [166, 47, 205] using the legitimate software, as we describe next. Consequently, the vTPM cannot be used directly to provide the runtime integrity of VMs.

Rollback Attack

The adversary dumps the state of the $TPM_{t=1}$ containing legitimate measurements (Figure 4.1 1). Then, she compromises the VM's integrity and restores the previously saved $TPM_{t=1}$ state. Although the integrity measurements stored in $TPM_{t=2}$ reflect the attack, the vTPM uses a legitimate $TPM_{t=1}$ state. This attack is feasible because the adversary has unrestricted control over the vTPM life-cycle and its memory, *i.e.*, she can copy files with the TPM state, spawn a new TPM instance, attach it to an arbitrary VM. We propose to protect against the rollback attack by tagging the vTPM state with the version number and the unique vTPM identifier. The recent version number is stored in the monotonic counter (MC), and the vTPM increases it before executing each non-idempotent operation. The vTPM protects its state from tampering by running inside the TEE. During the startup, the vTPM ensures that the MC value equals the version number read from the persistent state.

Man-in-the-middle Attacks

In the vTPM design, the hypervisor prepends a 4-byte vTPM identifier that allows routing the communication to the correct vTPM instance. However, the link between the vTPM and the VM is unprotected [47], and it is routed through an untrusted network. Consequently, an adversary can mount a masquerading attack to redirect the VM communication to an arbitrary vTPM (Figure 4.1 2a) by replacing the vTPM identifier inside the network package. To mitigate the attack, we propose to use the transport layer security (TLS) protocol [54] to protect the communication's integrity.

Although the TLS helps protect the communication's integrity, the lack of authentication between the vTPM and the hypervisor still enables an adversary to fully control the communication by mounting a man-in-the-middle (MitM) attack. In more detail, an adversary can configure the hypervisor in a way it communicates with vTPM via an intermediary software, which intercepts the communication (Figure 4.1 2b). She can then drop arbitrary measurements or perform the TPM reset attack [142], thus compromising the vTPM security guarantees.

To mitigate the attack, the vTPM must ensure the remote peer's integrity (is it the correct hypervisor?) and its locality (is the hypervisor running on the same platform?). Although the TEE local attestation gives information about software integrity and locality, we cannot use it here because the hypervisor cannot run inside the TEE. However, suppose we find a way to satisfy the locality condition. In that case, we can leverage runtime integrity measurements (IMA) to verify the hypervisor's integrity because, among trusted software running on the platform, there can be only software that connects to the vTPM—the hypervisor. To satisfy the locality condition, we make the following observation: Only software running on the same platform has direct access to the same hardware TPM. We propose to share a secret between the vTPM and the hypervisor using the hardware TPM (§4.4.3). The vTPM then authenticates the hypervisor by verifying that the hypervisor presents the secret in the pre-shared key TLS authentication.

Finally, an adversary who compromises the guest operating system can mount the cuckoo attack [205] to impersonate the legitimate VM. An adversary can modify the TPM driver inside a guest operating system to redirect the TPM communication to a remote TPM (Figure 4.1 2c). A verifier running inside a compromised VM cannot recognize if he communicates with the vTPM attached to his VM or with a remote vTPM attached to another VM. The verifier is helpless because he cannot establish a secure channel to the vTPM that would guarantee communication with the local vTPM. To mitigate the attack, we propose leveraging the TEE

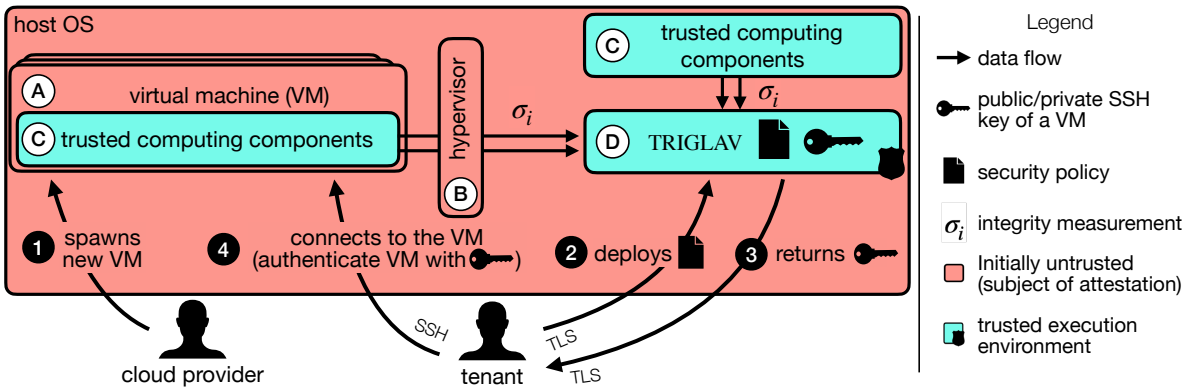


Figure 4.2. The high-level overview of TRIGLAV. The VM's SSH key is bound to the VM's integrity state defined in the policy.

attestation protocol to establish a secure communication channel between the verifier and the vTPM and to use it to exchange a secret allowing the verifier to identify the vTPM instance uniquely (§4.4.4).

4.4 TRIGLAV Design

Our objective is to provide an architecture that:

- protects legacy applications running inside a VM from threats defined in §4.2,
- requires zero-code changes to legacy applications and the VM setup,
- permits tenants to remotely attest to the execution environment's runtime integrity without possessing any vendor-specific hardware.

4.4.1 High-level Overview

Figure 4.2 shows an overview of the cloud node running TRIGLAV. It consists of the following four components:

(A) the VM,

(B) the hypervisor managing the VM, providing it with access to physical resources and isolating from other VMs,

(C) trusted computing components enabling hypervisor's runtime integrity enforcement and attestation,

(D) TRIGLAV, software executed inside TEE that allows tenants to attest and enforce the VMs' integrity.

The configuration, the execution, and the operation of the above components are subject to attestation. First, the cloud operator bootstraps the cloud node and starts TRIGLAV. At the tenant's request, the cloud provider spawns a VM (❶). Next, the tenant establishes trust with TRIGLAV (§4.4.4), which becomes the first trusted component on a remote computer. The tenant requests TRIGLAV to check if the hypervisor conforms to the policy (❷), which contains tenant-specific trust constraints, such as integrity measurements (§4.4.6). TRIGLAV uses IMA and TPM to verify that the computer's runtime integrity conforms to the policy and then generates a VM's public/private key pair. The public key is returned to the tenant (❸). TRIGLAV protects access to the private key, *i.e.*, it permits the VM to use the private key only

if the host and guest operating systems match the integrity defined inside the policy. Finally, the tenant establishes trust with the VM during the SSH-handshake. He verifies that the VM can use the private key corresponding to the previously obtained public key (❹). The tenant authenticates himself in a standard way, using his own SSH private key. His SSH public key is embedded inside the VM's image or provisioned during the VM deployment.

4.4.2 Platform Bootstrap

The cloud provider is responsible for the proper computer initialization. She must turn on support for hardware technologies (*i.e.*, TPM, DRTM, TEE), launch the hypervisor, and start TRIGLAV. The tenant ensures that the platform was correctly initialized when he establishes trust in the platform (§4.4.4).

First, TRIGLAV establishes a connection with the hardware TPM using the TPM attestation; it reads the TPM certificate and generates an attestation key following the *activation of credential* procedure ([12] p. 109-111). TRIGLAV ensures it communicates with the local TPM using a protocol detecting the TPM cuckoo attack introduced in chapter 3. Eventually, TRIGLAV reads the TPM quote, which certifies the DRTM launch and the measurements of the hypervisor's integrity.

4.4.3 VM Launch

The cloud provider requests the hypervisor to spawn a new VM. The hypervisor allocates the required resources and starts the VM providing it with TPM access. At the end of the process, the cloud provider shares the connection details with the tenant, allowing the tenant to connect to the VM.

TRIGLAV emulates multiple TPMs inside the TEE because many VMs cannot share a single hardware TPM [21]. When requested by the hypervisor, TRIGLAV spawns a new TPM instance accessible on a unique TCP port. The hypervisor connects to the emulated TPM and exposes it to the VM as a standard character device. We further use the term *emulated TPM* to describe a TEE-based TPM running inside the hypervisor and distinguish it from the software-based TPM proposed by the vTPM design.

The communication between the hypervisor and the emulated TPM is susceptible to MitM attacks (§4.3.2). Unlike TRIGLAV, the hypervisor does not execute inside the TEE, preventing TRIGLAV from using the TEE attestation to verify the hypervisor identity. However, TRIGLAV confirms the hypervisor identity by requesting it to present a secret when establishing a connection. TRIGLAV generates a secret inside the TEE and seals it to the hardware TPM via an encrypted channel ([92] §19.6.7). Only software running on the same operating system as TRIGLAV can unseal the secret. Thus, it is sufficient to check if only trusted software executes on the platform to verify that it is the legitimate hypervisor who presents the secret.

Figure 4.3 shows the procedure of attaching an emulated TPM to a VM. Before the hypervisor spawns a VM, it commands TRIGLAV to emulate a new software-based TPM (❶). TRIGLAV creates a new emulated TPM, generates a secret, and seals the secret with the hardware TPM (❷). TRIGLAV returns the TCP port and the sealed secret to the hypervisor. The hypervisor unseals the secret from the hardware TPM (❸) and establishes a TLS connection to the emulated TPM authenticating itself with the secret (❹). At this point, the hypervisor spawns a VM. The VM boots up, the firmware and IMA send integrity measurements to the emulated TPM (❺). To protect against the rollback attack, each integrity measurement causes the em-

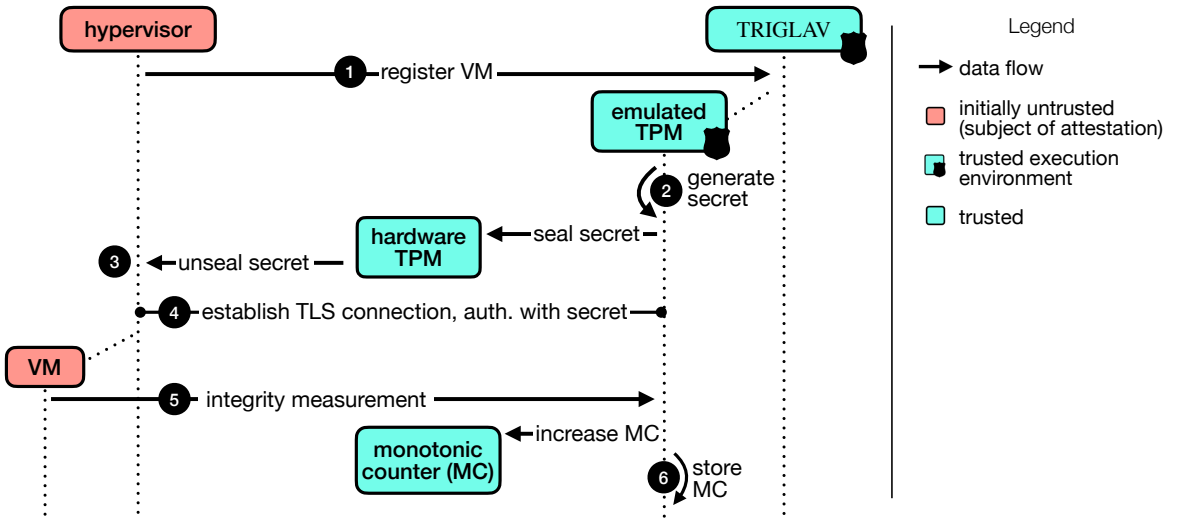


Figure 4.3. TRIGLAV emulates TPMs inside the TEE. To prevent the MitM attack, TRIGLAV authenticates the connecting hypervisor by sharing with him a secret via a hardware TPM. To mitigate the rollback attack, the emulated TPM increments the monotonic counter value on each non-idempotent command.

ulated TPM to increment the hardware-based MC and store the current MC value inside the emulated TPM memory (❻).

TRIGLAV permits only one client connection and does not permit reconnections to prevent the attachment of new VMs to the already initialized emulated TPM. Although an adversary might redirect the hypervisor to a fake emulated TPM exporting a false secret, such an attack is detected when establishing trust with the VM (§4.4.4).

4.4.4 Establishing Trust

The tenant establishes trust with the VM in three steps. First, he verifies that TRIGLAV executes inside the TEE and runs on genuine hardware (a CPU providing the TEE functionality). He then extends trust to the hypervisor and VM by leveraging TRIGLAV to verify and enforce the runtime integrity of the host and guest operating systems. Finally, he connects to the VM, ensuring it is the VM provisioned and controlled by TRIGLAV.

Since the TRIGLAV design does not restrict tenants to possess any vendor-specific hardware and the existing TEE attestation protocols are not standardized, we propose to add an extra level of indirection. Following the existing solutions [88], we rely on a trusted certificate authority (CA) that performs the TEE-specific attestation before signing an X.509 certificate confirming the TRIGLAV's integrity and the underlying hardware genuineness. The tenant establishes trust with TRIGLAV during the TLS-handshake, verifying that the presented certificate was issued to TRIGLAV by the CA.

Although the tenant remotely ensures that TRIGLAV is trusted, he has no guarantees that he connects to his VM controlled by TRIGLAV because the adversary can spoof the network [285] redirecting the tenant's connection to an arbitrary VM. To mitigate the threat, TRIGLAV generates a secret and shares it with the tenant and the VM. When the tenant establishes a connection, he uses the secret to authenticate the VM. Only the VM which integrity conforms to the policy has access to this secret.

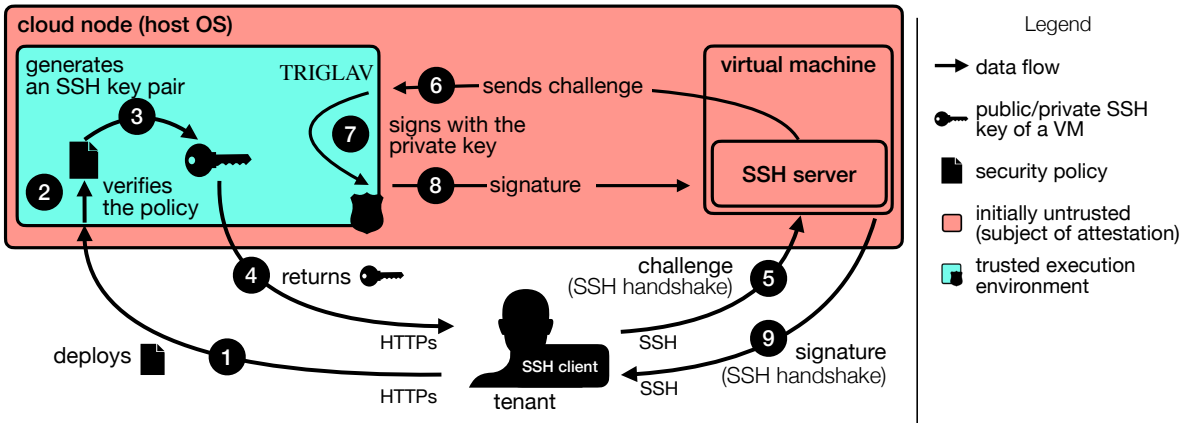


Figure 4.4. The high-level view of the attestation protocol. TRIGLAV generates an SSH public/private key pair inside the TEE. The tenant receives the public key as a result of the policy deployment. To mitigate the MitM attacks, the tenant challenges the VM to prove it has access to the private key. TRIGLAV signs the challenge on behalf of the VM if and only if the platform integrity conforms with the policy.

Figure 4.4 shows a high-level view of the protocol. First, the tenant establishes a TLS connection with TRIGLAV to deploy the policy (❶). TRIGLAV verifies the platform integrity against the policy (❷), and once succeeded, it generates the SSH key pair (❸). The public key is returned to the tenant (❹) while the private key remains inside the TEE. TRIGLAV enforces that only a guest operating system which runtime integrity conforms to the policy can use the private key for signing. Second, the tenant initializes an SSH connection to the VM, expecting the VM to prove the possession of the SSH private key. The SSH client requests the SSH server running inside the VM to sign a challenge (❺). The SSH server delegates the signing operation to TRIGLAV (❻). TRIGLAV signs the challenge using the private key (❼) if and only if the hypervisor's and VM's integrity match the policy. The SSH private key never leaves TRIGLAV; only a signature is returned to the SSH server (❽). The SSH client verifies the signature using the SSH public key obtained by the tenant from TRIGLAV (❾). The SSH server also authenticates the tenant, who proves his identity using his own private SSH key. The SSH server is configured to trust his SSH public key. The tenant established trust in the remote platform as soon as the SSH handshake succeeded.

4.4.5 Policy Enforcement

TRIGLAV policy enforcement mechanism guarantees that the VM runtime integrity conforms to the policy. At the host operating system, TRIGLAV relies on the IMA integrity-enforcement [101] to prevent the host kernel from loading to the memory files that are not digitally signed. Specifically, each file in the filesystem has a digital signature stored inside its extended attribute. IMA verifies the signature issued by the cloud provider before the kernel loads the file to the memory. The certificate required for signature verification is loaded from initramfs (measured by the DRTM) to the kernel's keyring. At the guest operating system, IMA inside the guest kernel requires the TRIGLAV approval to load a file to the memory. The emulated TPM, controlled by TRIGLAV, returns a failure when IMA tries to extend it with measurement not conforming to the policy. The failure instructs IMA not to load the file.

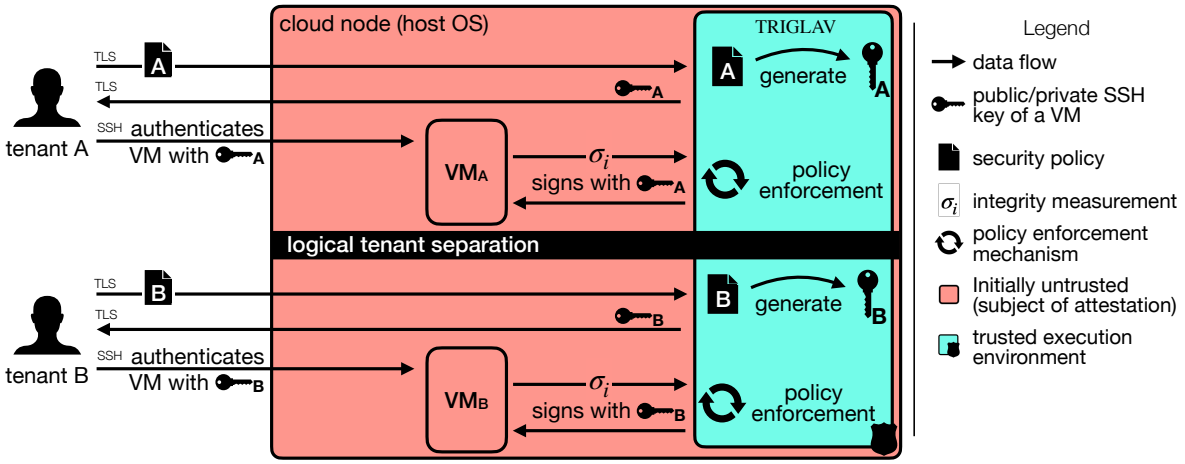


Figure 4.5. Multiple tenants interacting with TRIGLAV concurrently. TRIGLAV generates a dedicated SSH key for each deployed security policy and allows using it only if the VM's integrity conforms to the security policy.

4.4.6 Tenant Isolation and Security Policy

Multiple applications with different security requirements might coexist on the same physical computer. TRIGLAV allows ensuring that applications run in isolation from each other and match their security requirements. Figure 4.5 shows how TRIGLAV assigns each VM a pair of a public and private key. The keys are bound with the application's policy and the VM's integrity. Each tenant uses the public key to verify that he interacts with his VM controlled by the integrity-enforced hypervisor.

Listing 4.1: Example of the TRIGLAV's security policy

```
host:
  tpm: |-
    -----BEGIN CERTIFICATE-----
    # Manufacturer CA certificate
    -----END CERTIFICATE-----
  drtm:  # measurements provided by the DRTM
    - {id: 17, sha256: f9ad0...cb}
    - {id: 18, sha256: c2c1a...c1}
    - {id: 19, sha256: a18e7...00}
  certificate: |-
      -----BEGIN CERTIFICATE-----
      # software update certificate
      -----END CERTIFICATE-----
guest:
  enforcement: true
  pcrs:  # boot measurements (e.g., secure boot)
    - {id: 0, sha256: a1a1f...dd}
  measurements:  # legal integrity measurement digests
    - "e0a11...2a" # SHA digest over a startup script
    - "3a10b...bb" # SHA digest over a library
  certificate:
```

```
22      - |-
23          -----BEGIN CERTIFICATE-----
24          # certificate of the signer, e.g., OS updates
25          -----END CERTIFICATE-----
26          -----BEGIN CERTIFICATE-----
27          # software update certificate
28          -----END CERTIFICATE-----
```

Listing 4.1 shows an example of a security policy. The policy is a text file containing a whitelist of the hardware TPM manufacturer's certificate chain (line 4), DRTM integrity measurements of the host kernel (lines 6-9), integrity measurements of the guest kernel (line 16), and runtime integrity measurements of the guest operating system (lines 18-20, 24). The certificate chain is used to establish trust in the underlying hardware TPM. TRIGLAV compares DRTM integrity measurements with PCR values certified by the TPM to ensure the correct hypervisor with enabled integrity-enforced mechanism was loaded. TRIGLAV uses runtime integrity measurements to verify that only expected files and software have been loaded to the guest operating system memory. A dedicated certificate (line 24) makes the system scalable because it permits more files to be loaded to the memory without redeploying the policy. Specifically, it is enough to sign the software, which we allow to execute, with the corresponding private key to make the software pass through the integrity-enforcement mechanism. Similarly, dedicated certificates (lines 12, 27) allow for software updates of both host and guest operating system thanks to a dedicated *trusted software repository* discussed in detail in chapter 6.

4.5 Implementation

4.5.1 Technology Stack

We decided to base the prototype implementation on the Linux kernel because it is an open-source project supporting a wide range of hardware and software technologies. It is commonly used in the cloud and, as such, can demonstrate the practicality of the proposed design. QEMU [17] and kernel-based virtual machine (KVM) [147] permit to use it as a hypervisor. We rely on Linux IMA [225] as an integrity enforcement and auditing mechanism built-in the Linux kernel.

We chose Alpine Linux because it is designed for security and simplicity in contrast to other Linux distributions. It consists of a minimum amount of software required to provide a fully functional operating system that permits keeping a trusted computing base (TCB) low. All userspace binaries are compiled as position-independent executables with stack-smashing protection and relocation read-only memory corruption mitigation techniques. Those techniques help mitigate the consequences of, for example, buffer overflow attacks that might lead to privilege escalation or arbitrary code execution. To restrict the host from accessing guest memory and state, we follow existing security-oriented commercial solutions [112] that disable certain hypervisor features, such as hypervisor-initiated memory dump, huge memory pages on the host, memory swapping, memory ballooning through a virtio-balloon device, and crash dumps. For production implementations, we propose to rely on microkernels like formally proved seL4 [148].

We rely on SGX as the TEE technology. The SGX remote attestation [133] allows us to

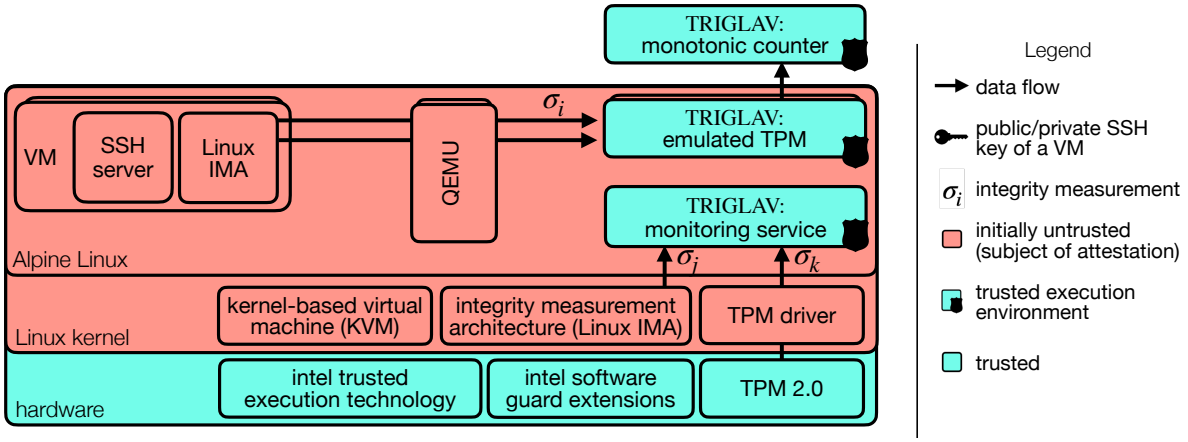


Figure 4.6. The overview of the TRIGLAV prototype implementation.

verify if the application executes inside an enclave on a genuine Intel CPU. We implemented TRIGLAV in Rust [180], which preserves memory-safety and type-safety. To run TRIGLAV inside an SGX enclave, we used the SCONE framework [11] and its Rust cross-compiler. We also exploited the Intel trusted execution technology (TXT) [87] as a DRTM technology because it is widely available on Intel CPUs. We used the open-source software tboot [127] as a pre-kernel bootloader that establishes the DRTM with TXT to provide the measured boot of the Linux kernel.

4.5.2 Prototype Architecture

The TRIGLAV prototype architecture consists of three components: the monitoring service, the emulated TPM, and the monotonic counter service. All the components execute inside an SGX enclave.

The monitoring service is the component that leverages Linux IMA and the hardware TPM to collect integrity measurements of the host operating system. There is one monitoring service running on the host operating system. It is available on a well-known port on which it exposes a TLS-protected REST API used by tenants to deploy the policy. We based this part of the implementation on the CHORS's agent, discussed in chapter 3, that provides a mechanism to detect the TPM locality. The monitoring service spawns emulated TPMs and intermediates in the secret exchange between QEMU and the emulated TPM. Specifically, it generates and seals to the hardware TPM the secret required to authenticate the QEMU process, and passes this secret to an emulated TPM.

The emulated TPM is a software-based TPM emulator based on the libtpms library [19]. It exposes a TLS-based API allowing QEMU to connect. The connection is authenticated using the secret generated inside an SGX enclave and known only to processes that gained access to the hardware TPM. We extracted the emulated TPM into a separate component because of the libtpms implementation, which requires running each emulator in a separate process.

The monotonic counter service (MCS) provides access to a hardware monotonic counter (MC). Emulated TPMs use it to protect against rollback attacks. We designed the MCS as a separate module because we anticipate that due to hardware MC limitations (*i.e.*, high latency, the limited number of memory overwrites [242]), a distributed version of the MCS, *e.g.*, ROTE [179], might be required. However, the MCS might also be deployed locally to leverage built-in SGX MC [36] accessible on the same platform where the monitoring service and

emulated TPM run.

4.5.3 Monotonic Counter Service

We implemented a monotonic counter service (MCS) as a service executed inside the SGX enclave. It leverages the high-endurance indices defined by the TPM 2.0 specification [92] to provide the MC functionality. The MCS relies on the TPM attestation to establish trust with the TPM chip offering hardware MC, and on the encrypted and authenticated communication channel ([92] §19.6.7) to protect the integrity and confidentiality of the communication with the TPM chip from the enclave. The MCS exposes a REST API over a TLS (§4.5.4), allowing other enclaves to increment and read hardware monotonic counters remotely.

The emulated TPM establishes trust with the MCS via the TLS-based SGX attestation (§4.5.4) and maintains the TLS connection open until the emulated TPM is shutdown. We implemented the emulated TPM to increase the MC before executing any non-idempotent TPM command, *e.g.*, extending PCRs, generating keys, writing to non-volatile memory. The MC value and the TLS credentials are persisted in the emulated TPM state, which is protected by the SGX during runtime and at rest via sealing. When the emulated TPM starts, it reads the MC value from the MCS and then checks the emulated TPM state freshness by verifying that its MC value equals the value read from the MCS.

4.5.4 TLS-based SGX Attestation

We use the SCONE key management system (CAS) [258] to perform remote attestation of TRIGLAV components, verify SGX quotes using Intel attestation service (IAS) [9], generate TLS credentials, and distribute the credentials and the CAS CA certificate to each component during initialization. TRIGLAV components are configured to establish mutual authentication over TLS, where both peers present a certificate, signed by the same CAS CA, containing an enclave integrity measurement. Tenants do not perform the SGX remote attestation to verify the monitoring service identity and integrity. Instead, they verify the certificate exposed by a remote peer during the policy deployment when establishing a TLS connection to the monitoring service. In our prototype implementation, we force tenants to trust CAS. The production implementation might use Intel SGX-RA [149] to achieve similar functionality without relying on an external key management system.

4.5.5 VM Integrity Enforcement

The current Linux IMA implementation extends the integrity digest of the IMA log entry to all active TPM PCR banks. For example, when there are two active PCR banks (*e.g.*, SHA-1 and SHA-256), both are extended with the same value. We decided to make a minor modification in the Linux kernel, which permitted us to share with the emulated TPM not only the integrity digest but also the file's measurement and the file's signature. We modified the content of the *PCR_Extend* command sent by the Linux IMA in a way it uses the SHA-1 bank to transfer the integrity digest, the SHA-256 bank to transfer the file's measurement digest, and the SHA-512 bank to transfer the file's signature. In the emulated TPM, we intercept the PCR_extend command to extract the file's measurement and the file's digest. We use obtained information to enforce the policy; if the file is not permitted to be executed, the

emulated TPM process closes the TLS connection, which is a signal to the QEMU process to shut down the VM.

4.5.6 SSH Integration

To enable a secure connection to the VM, we relied on the OpenSSH server. It supports the PKCS#11 [197] standard, which defines how to communicate with cryptographic devices, like TPM, to perform cryptographic operations using the key without retrieving it from the TPM.

We configured an OpenSSH server running inside the guest operating system to use an SSH key stored inside the emulated TPM running on the host operating system. Importantly, the VM's SSH private key is generated and stored inside the SGX enclave, and it never leaves it. The SSH server, via PKCS#11, uses it for the TLS connection only when TRIGLAV authorizes access to it. The tenant uses his own SSH private key, which is not managed by TRIGLAV.

4.6 Evaluation

In this chapter, we answer the question if TRIGLAV is practical to protect legacy applications.

Testbed. Experiments execute on a Dell PowerEdge R330 servers equipped with an Intel Xeon E3-1270 v5 CPU, 64 GiB of RAM, Infineon 9665 TPM 2.0 discrete TPM chips (dTPMs). Experiments using an integrated TPM (iTPM) run on Intel NUC7i7BNH machine, which has the Intel platform trusted technology (PTT) running on Intel ME 11.8.50.3425 powered by Intel Core i7-7567U CPU and 8 GiB of RAM.

All machines have a 10 Gb Ethernet network interface card (NIC) connected to a 20 Gb/s switched network. The SGX, TXT, TPM 2.0, Intel virtualization technology for directed I/O (VT-d), and single root input/output virtualization (SR-IOV) technologies are turned on in the UEFI/BIOS system configuration. The hyper-threading is switched off. The enclave page cache (EPC) is configured to reserve 128 MiB of RAM.

On host and guest operating systems, we run Alpine 3.10 with Linux kernel 4.19. We modified the guest operating system kernel according to the description in §4.5.5. We adjusted quick emulator (QEMU) 3.1.0 to support TLS-based communication with the emulated TPM as described in §4.4.3.

4.6.1 Micro-benchmarks

Are TPM monotonic counters practical to handle the rollback protection mechanism?

Strackx and Piessens [242] reported that the TPM 1.2 memory gets corrupted after a maximum of 1.450M writes and has a limited increment rate (one increment per 5 sec). We run an experiment to confirm or undermine the hypothesis that those limitations apply to the TPM 2.0 chip. We were continuously incrementing the monotonic counter in dTPM and iTPM chips. The dTPM chip reached 85M increments, and it did not throttle its speed. The iTPM chip slowed down after 7.3M increments limiting the increment latency to 5 sec. We did not observe any problem with the TPM memory.

Table 4.1. The latency of main operations in the TPM-based MCS. σ states for standard deviation.

	Read	Increase
discrete TPM	42 ms ($\sigma = 2$ ms)	40 ms ($\sigma = 2$ ms)
integrated TPM	25 ms ($\sigma = 2$ ms)	32 ms ($\sigma = 1$ ms)

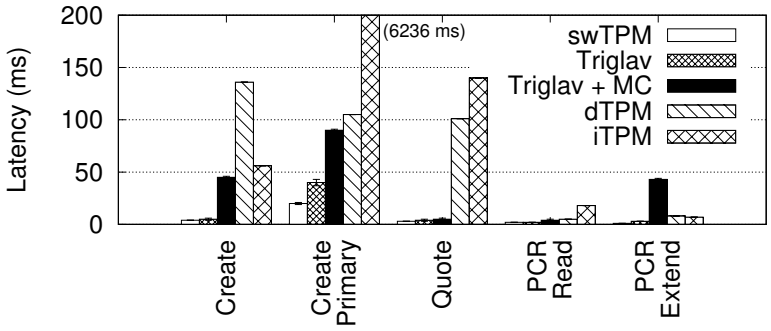


Figure 4.7. TPM operations latency depending on the TPM.

What is the cost of the rollback protection mechanism?

Each non-idempotent TPM operation causes the emulated TPM to communicate with the MCS and might directly influence the TRIGLAV performance. We measured the latency of the TPM-based MCS *read* and *increment* operations. In this experiment, the MCS and the test client execute inside an SGX enclave. Before the experiment, the test client running on the same machine establishes a TLS connection with the MCS. The connection is maintained during the entire experiment to keep the communication overhead minimal. The evaluation consists of sending 5k requests and measuring the mean latency of the MCS response.

Table 4.1 shows that the MCS using iTPM performs from 1.25× to 1.68× faster than its version using dTPM. The read operation on the iTPM is faster than the increment operation (25 ms versus 32 ms, respectively). Differently, on dTPM both operations take a similar amount of time (about 40 ms).

What is the cost of running the TPM emulator inside TEE and with the rollback protection mechanism? Is it slower than a hardware TPM used by the host operating system?

As a reference point to evaluate the emulated TPM's performance, we measured the latency of various TPM commands executed in different implementations of TPMs. The TPM quotes were generated with the elliptic curve digital signature algorithm (ECDSA) using the P-256 curve and SHA-256 bit key. PCRs were extended using the SHA-256 algorithm.

Figure 4.7 shows that except for the PCR extend operation, the SGX-based TPM with rollback protection is from 1.2× to 69× faster than hardware TPMs and up to 6× slower than the unprotected software-based swTPM. Except for the create primary command, which derives a new key from the TPM seed, we did not observe performance degradation when running the TPM emulator inside an enclave. However, when running with the rollback protection, the TPM slows down the processing of non-idempotent commands (*e.g.*, PCR_Extend) due to the additional time required to increase the MC.

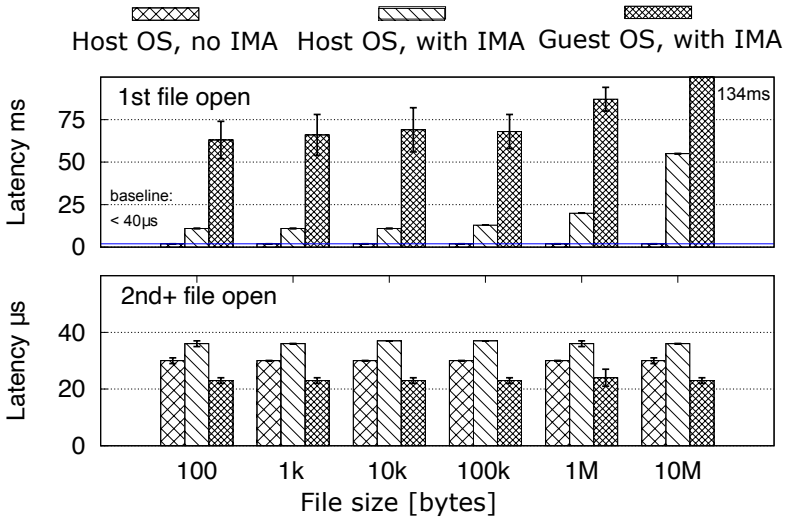


Figure 4.8. File opening times with and without Linux IMA.

How much IMA impacts file opening times?

Before the kernel executes the software, it verifies if executable, related configuration files, and required dynamic libraries can be loaded to the memory. The IMA calculates a cryptographic hash over each file (its entire content) and sends the hash to the TPM. We measure how much this process impacts the opening time of files depending on their size.

Figure 4.8 shows that the IMA inside the guest operating system incurs higher overhead than the IMA inside the host operating system. It is primarily caused by (i) the higher latency of the TPM extend command ($\sim$ 43 ms) that is dominated by a slow network-based monotonic counter, (ii) the IMA mechanism itself that has to calculate the cryptographic hash over the entire file even if only a small part of the file is actually read, and (iii) the less efficient data storage used by the VM (virtualized storage, QCOW format).

In both systems, the IMA takes less than 70 ms when loading files smaller than 1 MB (99% of files in the deployed prototype are smaller than 1 MB). Importantly, IMA measures the file only once unless it changes. Figure 4.8 shows that the next file reads take less than 40 μs regardless of the file size.

4.6.2 Macro-benchmarks

We run macro-benchmarks to measure performance degradation when protecting popular applications, *i.e.*, the nginx web-server [65] and the memcached cache system [43], with TRIGLAV. We compare the performance of four variants for each application running on the host operating system (native), inside a SCONE-protected Docker container on the host operating system (SCONE), inside a guest operating system (VM), inside a TRIGLAV-protected guest operating system with rollback protection turned on (TRIGLAV). Please note that TRIGLAV operates under a weaker threat model than SCONE. We compare both systems to show the tradeoff between the security, performance, and the required resources.

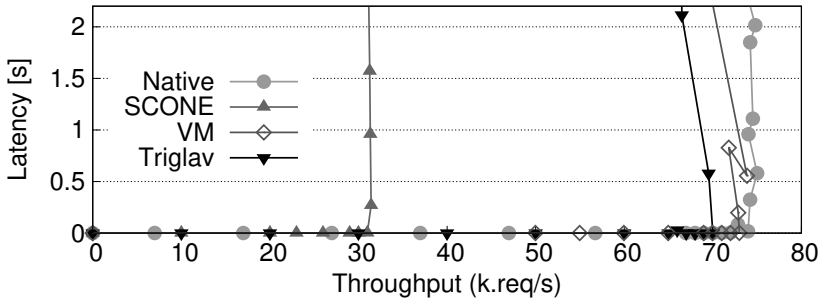


Figure 4.9. Throughput/latency of the nginx.

How much does TRIGLAV influence the throughput of a web server, *e.g.*, nginx?

We configured nginx to run a single worker thread with turned off gzip compression and logging, according to available SCONE's benchmark settings. Then, we used wrk2 [248], running on a different physical host, to simulate 16 clients (4 per physical core) concurrently fetching a pre-generated 10 KiB binary uncompressed file for 45 s.[10] We were increasing the frequency of the fetching until the response times started to degrade. Except for the reference measurement (native) run on the bare metal, nginx run inside a VM with access to all available cores and 4 GB of memory.

Figure 4.9 shows that TRIGLAV achieved 0.94× of the native throughput, reaching 70k requests. The SCONE variant reached about 31k requests, which is 0.45× of the TRIGLAV throughput. We observed low-performance degradation incurred by the virtualization (less than 2%). The TRIGLAV overhead is caused mostly by the IMA.

Does TRIGLAV influence the throughput of systems that extensively use in-memory storage, *i.e.*, memcached?

In this experiment, we used memtier [216] to generate load by sending *GET* and *SET* requests (at 1:1 ratio) of 500 bytes of random data to a memcached instance running on a different physical host. We calculated the memcached performance by computing the mean throughput achieved by the experiment before the throughput started to degrade (latency

[10]The limited network bandwidth dictated the file size—for larger sizes, we saturated the NIC bandwidth.

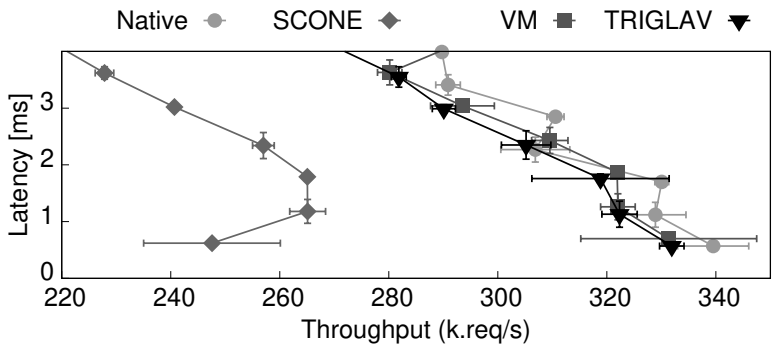


Figure 4.10. Throughput/latency of memcached.

lower than 2 ms). Except for the reference measurement (native) run on the bare metal, memcached run inside a VM with access to all available cores and 4 GB of memory.

Figure 4.10 presents how TRIGLAV influences the throughput-latency ratio of memcached. We observed small performance degradation when running memcached inside a VM. TRIGLAV achieved 0.98× of the native throughput. It is a result of how Linux IMA is implemented. During the memcached start, IMA measures the integrity of the memcached executable, dynamic libraries, and configuration files. But, it does not measure any data directly written to the memory during runtime. TRIGLAV throughput was 1.23× higher than the memcached run inside SCONE.

Table 4.2. The VM boot time depending on the TPM.

	MC	TPM	IMA	Boot time
no TPM	✗	✗	✗	9.7 sec ($\sigma = 0.1$ sec)
swTPM	✗	✓	✓	14.0 sec ($\sigma = 0.2$ sec)
TRIGLAV				
no MC	✗	✓	✓	14.1 sec ($\sigma = 0.3$ sec)
with MC	✓	✓	✓	50.8 sec ($\sigma = 0.4$ sec)
fast MC	✓	✓	✓	15.8 sec (estimate)

How the measured boot increases the VM boot time?

Table 4.2 shows how TRIGLAV impacts VM boot times. As a reference, we measure the boot time of a VM without any TPM attached. Then, we run experiments in which a VM has access to different implementations of a software-based TPMs. Except for the reference measurement, the Linux IMA is always turned on. Each VM has access to all available cores and 4 GB of memory. As the guest operating system, we run Ubuntu 18.10, a Linux distribution with a pre-installed tool (systemd-analyze) to calculate system boot times.

The measured boot increases the VM load time. It is caused by the IMA module that measures files required to initialize the operating system. We did not observe any difference in boot time between the setup with the swTPM [20] and the Triglav emulated TPM (*Triglav no MC*). However, when running the emulated TPM with the rollback protection (*Triglav with MC*), the VM boot time is 5.2× and 3.6× higher when compared to the reference and the swTPM setting, respectively. Alternative implementations of MCS, such as ROTE [179], offer much faster MC increments (1–2 ms) than the presented TPM-based prototype. We estimated that using TRIGLAV with a *fast MC* would slow down VM boot time only by 1.13×.

Does TRIGLAV incur performance degradation when multiple VMs are spawned?

We examine the scalability by running memcached concurrently in several VMs with and without TRIGLAV, *i.e.*, the *native* indicates memcached instances executing inside VMs whose integrity is neither measured nor enforced. Specifically, we calculate the performance degradation between the variant with and without TRIGLAV. *I.e.*, we do not compare the performance degradation between different numbers of VMs, because it depends on the limited amount of shared network bandwidth. In the scalability experiments, we assigned one physical core and 1 GB of RAM to each VM.

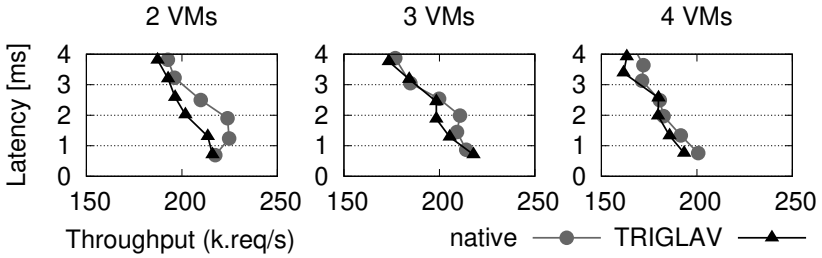


Figure 4.11. Throughput/latency of memcached depending on the number of concurrently executed VMs.

Figure 4.11 shows that when multiple VMs are concurrently running on the host operating system, TRIGLAV achieves 0.96×–0.97× of the native throughput.

4.7 Discussion

4.7.1 Alternative TEEs

The TRIGLAV design (§4.4) requires a TEE that offers a remote attestation protocol and provides confidentiality and integrity guarantees of TRIGLAV components executing in the host operating system. Therefore, the SGX used to build the TRIGLAV prototype (§4.5) might be replaced with other TEEs. In particular, TRIGLAV implementation might leverage Sanctum [46], Keystone [169], Flicker [184], or L4Re [217] as an SGX replacement. TRIGLAV might also leverage ARM TrustZone [172] by running TRIGLAV components in the *secure world* and exploiting the TPM attestation to prove its integrity.

4.7.2 Hardware-enforced VM Isolation

Hardware CPU extensions, such as SEV [116], Intel multi-key total memory encryption (MK-TME) [122], Intel trust domain extensions (TDX) [123], are largely complementary to the TRIGLAV design. They might enrich TRIGLAV design by providing the confidentiality of the code and data against rogue operators with physical access to the machine, compromised hypervisor, or malicious co-tenants. They also consider untrusted hypervisor excluding it from the TRIGLAV TCB. On the other hand, TRIGLAV complements these technologies by offering means to verify and enforce the runtime integrity of guest operating systems — the functionality easily available for bare-metal machines (via a hardware TPM) but not for virtual machines.

4.7.3 Trusted Computing Base

The prototype builds on top of software commonly used in the cloud (*i.e.*, Linux kernel, QEMU), which has a large TCB because it supports different processor architectures and hardware. TRIGLAV might be combined with other TEE and hardware extensions, resulting in a lower TCB and stronger security guarantees. Specifically, TRIGLAV could be implemented on top of a microkernel architecture, such as formally verified seL4 [148, 209], that provides stronger isolation between processes and a much lower code base (less than 10k source lines of code (SLOC) [148]), when compared to the Linux kernel. Compared to the prototype,

QEMU might be replaced with Firecracker [8], a virtual machine monitor written in a type-safe programming language that consists of 46k SLOC (0.16× of QEMU source code size) and is used in production by Amazon AWS cloud. The TCB of the prototype implementation might be reduced by removing superfluous code and dependencies. For example, most of the TPM emulator functionalities could be removed following the approach of μTPM [183]. TRIGLAV API could be built on top of the socket layer, allowing removal of HTTP dependencies that constitute 41% of the prototype implementation code.

4.7.4 Integrity Measurements Management

The policy composed of digests is sensitive to software updates because newer software versions result in different measurement digests. Consequently, any software update of an integrity-enforced system would require a policy update, which is impractical. Instead, TRIGLAV supports dedicated *update mirrors* serving updates containing digitally signed integrity measurements, a solution introduced and discussed in chapter 6.

Other measurements defined in the policy can be obtained from the national software reference library [198] or directly from the IMA-log read from a machine executed in a trusted environment, *e.g.*, development environment running on tenant premises. The amount of runtime IMA measurements can be further reduced by taking into account processes interaction to exclude some mutable files from the measurement [130, 226].

4.8 Related Work

VM attestation is a long-standing research objective. The existing approaches vary from VMs monitoring systems that focus on system behavior verification [99, 223, 208], intrusion detection systems [135, 224], or verifying the integrity of the executing software [74, 22]. TRIGLAV focuses on the VM runtime integrity attestation.

Following Terra [74] architecture, TRIGLAV leverages VMs to provide isolated execution environments constrained with different security requirements defined in a policy. Like Scalable Attestation [22], TRIGLAV uses software-based TPM to collect VM integrity measurements. Additionally, TRIGLAV extends the software-based TPM functionality by enforcing the policy and binding the attestation result with the VM connection, as proposed by IVP [230]. Unlike the idea of linking the remote attestation quote to the TLS certificate [84], TRIGLAV relies on the TEE to restrict access to the private key based on the attestation result. Following TrustVisor [183], TRIGLAV exposes trusted computing methods to legacy applications by providing them with dedicated TPM functionalities emulated inside the hypervisor. Unlike other works, TRIGLAV addresses the TPM remote attack (recall §4.3.2) at the VM level by combining integrity enforcement with key management and with the TEE-based remote attestation. Alternative approaches to TPM virtualization exist [243, 83]. However, the cuckoo attack remains the main problem. Moreover, the trusted hypervisor is still required to protect the TPM state and bind VMs with correct TPMs. In TRIGLAV, we enhanced the vTPM design [21] mostly because of the simplicity; no need for hardware [243] or the TPM specification [83] changes.

Hardware solutions, such as SEV [116], TDX [123], emerged to isolate VMs from the untrusted hypervisor and the cloud administrator. However, they lack the VM runtime integrity attestation, a key feature provided by TRIGLAV. TRIGLAV is complementary to them. Combin-

ing these technologies allows for better isolation of VM from the hypervisor and the administrator and for runtime integrity guarantees during the VM's runtime.

4.9 Summary

This chaper presented TRIGLAV, the VM attestation protocol allowing for verification that security-sensitive applications execute in the VM composed and controlled by expected software in expected configuration. TRIGLAV provides transparent support for legacy applications and requires no changes in the VM configuration. TRIGLAV also permits tenants to remotely attest to the platform runtime integrity without possessing any vendor-specific hardware by binding the VM integrity attestation with the SSH connection. Our evaluation shows that TRIGLAV is practical and incurs low performance overhead ($\leq$ 6%).

5 Secure Multi-Stakeholder Machine Learning Framework with GPU Support

5.1 Problem Statement

Machine learning (ML) techniques are widely adopted to build functional artificial intelligence (AI) systems. For example, face recognition systems allow paying at supermarkets without typing passwords; natural language processing systems allow translating information boards in foreign countries using smartphones; medical expert systems help to detect diseases at an early stage; image recognition systems help autonomous cars to identify road trajectory and traffic hazards. To build such systems, multiple parties or stakeholders with domain knowledge from various science and technology fields must cooperate since machine learning is fundamentally a multi-stakeholder computation, as shown in Figure 5.1. They would benefit from sharing their intellectual property – private training data, source code, and models – to jointly perform machine learning computations only if they can ensure their intellectual property remains confidential.

Training data owner. ML systems rely on training data to build *inference models*. However, the data is frequently sensitive and cannot be easily shared between disjoint entities. For example, healthcare data used for training diagnostic models contain privacy-sensitive patient information. The strict data regulations, such as general data protection regulation (GDPR) [63], impose an obligation on secure data processing. Specifically, the training data must be under the training data owner's control and must be protected while at rest, during transmission, and training computation.

Training code owner. The training code owner implements a *training algorithm* that trains an *inference model* over the training data. The *training code* (*e.g.*, Python code) typically contains an optimized training model architecture and tuned parameters that build the business value and the inference model quality. Thus, the training code is considered as confidential as the training data. The training requires high computing power, and, as such, it is economically justifiable to delegate its execution to the cloud. However, in the cloud, users with administrative access can easily read the training service source code implemented in popular programming languages, such as Python.

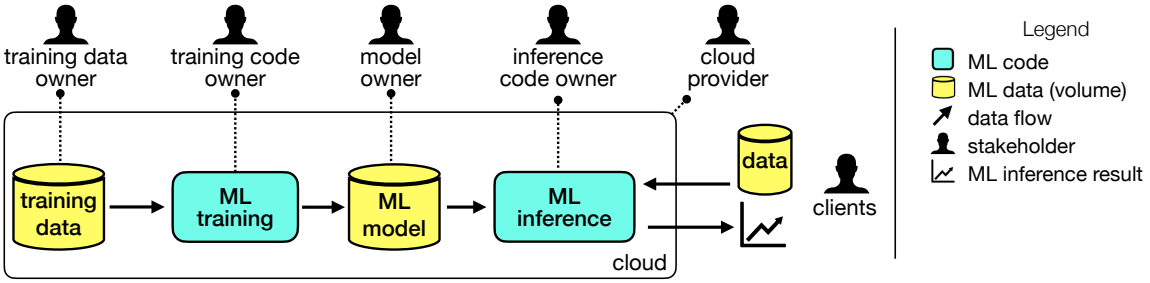


Figure 5.1. Stakeholders share source code, data, and computing power to build a ML application. They need a framework to establish mutual trust and share code and data securely.

Model owner. The inference model is the heart of any inference service. It is created by training the model with a large amount of training data. This process requires extensive computing power and is time-consuming and expensive. Thus, the model owner, a training code owner, or a third party that buys the model, must protect the model's confidentiality. The trained models may reveal the privacy of the training data [2]. Several works [70, 2] demonstrated that extracted images from a face recognition system look suspiciously similar to images from the underlying training data.

Inference code owner. The inference code is an AI service allowing clients to use the inference model on a business basis. The inference code is frequently developed using Python or JavaScript and hosted in the cloud. Thus, the confidentiality of the code and the integrity of the computation must be protected against an adversary controlling computing systems executing the AI service.

Inference data owner. The inference data owner is a client of an AI service. He wants to protect his input data. Imagine a person sending an X-ray scan of her brain to a diagnostic service to check for a brain tumor. The inference data, *e.g.*, a brain's scan, is privacy-sensitive and must not be accessible by the AI service provider.

To build an AI service, stakeholders must trust that others follow the rules protecting each other's intellectual property. However, it is difficult to establish trust among them. First, some stakeholders might collude to gain advantages over others [196]. Second, even a trustworthy stakeholder might lack expertise in protecting their intellectual property from a skilled attacker gaining access to its computing resources [218, 57]. We tackle the following problem: How to allow stakeholders to jointly perform machine learning to unlock all AI benefits without revealing their intellectual property?

Recent works [187, 160] demonstrated that cryptographic techniques, such as secure multi-party computation [278] and fully homomorphic encryption [78], incur a large performance overhead, which currently prevents their adoption for computing-intensive ML. Alternative approaches [199, 167] adopted trusted execution environments (TEEs) [185] to build ML systems showing that TEEs offer orders of magnitude faster ML computation, at the cost of weaker security guarantees compared to pure cryptographic solutions. Specifically, the pure cryptographic solutions compute on encrypted data, while in TEE-based approaches, a trusted ML software processes the plaintext data in a CPU-established execution environment (called *enclave*), which is isolated from the untrusted operating system and administrator. Although promising for the ML inference, TEEs still incur considerable performance overhead for memory-intensive computations, like deep training, because of the limited memory accessible to the enclave and lack of support for hardware accelerators, like graphical processing units (GPUs). Thus, since TEEs alone are not enough for the ML training

processes, we raise the question: What trade-off between security and performance has to be made to allow the ML training to access hardware accelerators?

5.2 Contribution

We propose PERUN, a framework allowing stakeholders to share their code and data only with certain ML applications running inside an enclave and on a trusted operating system. PERUN relies on encryption to protect the intellectual property and on a trusted key management service to generate and distribute the corresponding cryptographic keys. TEE provides confidentiality and integrity guarantees to ML applications and to the key management service. Trusted computing techniques [73] provide integrity guarantees to the operating system, allowing ML computations to access hardware accelerators. Our evaluation shows that PERUN achieves 0.96× of native performance execution on the GPU and a speedup of up to 1560× in training a real-world medical dataset compared to a pure TEE-based approach [167].

To summarize, in this chapter, we make the following contributions: (i) We designed a secure multi-stakeholder ML framework that allows stakeholders to cooperate while protecting their intellectual property (§5.4.1, §5.4.2)) and select trade-off between the security and performance, allowing for hardware accelerators usage (§5.4.3,§5.4.4). (ii) We implemented PERUN prototype (§5.5) and evaluated it using real-world datasets (§5.6).

5.3 Threat Model

Stakeholders are financially motivated businesses that cooperate to perform ML computation. Each stakeholder delivers an input (*e.g.*, input training data, code, and ML models) as its intellectual property for ML computations. The intellectual property must remain confidential during ML computations. The stakeholders have limited trust. They do not share their intellectual property directly, but they encrypt them so that only other stakeholders' applications, which source code they can inspect under a non-disclosure agreement or execute in a sandbox, can access the encryption key to decrypt it.

An adversary wants to steal a stakeholder's intellectual property when it resides on a computer executing ML computation. Such a computer might be provisioned in the cloud or a stakeholder's data center, *e.g.*, a hybrid cloud model. In both cases, an adversary has no physical access to the computer. For this, we rely on state-of-the-art practices controlling and restricting access to the data center to trusted entities.

However, an adversary might exploit an operating system misconfiguration or use social engineering to connect to the operating system remotely. We assume she can execute privileged software to read an ML process's memory after getting administrative access to the operating system executing ML computation. One of the mitigation techniques used in PERUN, integrity measurement architecture (IMA) [225], effectively limits software that can execute on the computer under the assumption that this software, which is considered trusted, behaves legitimately also after it has been loaded to the memory, *i.e.*, an adversary cannot tamper with the process' code after it has been loaded to the memory. This might be achieved using existing techniques, like enforcing control flow integrity [143], fuzzing [281], formally proving the software implementation [287], using memory-safe languages [180], using memory corruption mitigation techniques, like position-independent executables, stack-smashing protection, relocation read-only techniques, or others.

The CPU with its hardware features, hardware accelerators, and secure elements (*e.g.*, TPM) are trusted. We exclude micro-architectural and side-channel attacks, like Foreshadow [260] or Spectre [151]. We rely on the soundness of the cryptographic primitives used within software and hardware components.

5.4 Design

Our objective is to provide an architecture that:

- supports multi-stakeholder ML computation,
- requires zero code changes to the existing ML code,
- allows for a trade-off between security and performance,
- uses hardware accelerators for computationally-intensive tasks.

5.4.1 High-level Overview

Figure 5.2 shows the PERUN framework architecture that supports multi-stakeholder computation and the use of dedicated hardware accelerators. The framework consists of five components: (i) *stakeholders*, the parties who want to perform ML jointly while keeping their intellectual property protected; (ii) *security policy manager*, a key management and configuration service that allows stakeholders to share intellectual properties for ML computations without revealing them; (iii) *ML computation* including training and inference; (iv) *GPU*, hardware accelerators enabling high-performance ML computation; and (v) *TEE* and *TPM*, *secure elements* enabling confidentiality and integrity of ML computations on untrusted computing resources.

To allow multiple stakeholders to perform ML and keep their intellectual property confidential, we propose that the intellectual property remains under the stakeholder's control. To realize that idea, we design the security policy manager that plays the role of the *root of trust*. Stakeholders establish trust in this component using the *remote attestation* mechanism, like [133], offered by a TEE. The TEE, *e.g.*, Intel software guard extensions (SGX) [45], guarantees the confidentiality and integrity of processed code and data. After stakeholders ensure the security policy manager executes in the TEE, they submit security policies defining access control to their encryption keys. Each stakeholder's intellectual property is encrypted with a different key, and the security policy manager uses security policies to decide who can access which keys. From a technical perspective, the security policy manager generates the keys inside the TEE and sends them only to authenticated ML computations executing inside the TEE. Thus, these keys cannot be seen by any human.

Depending on individual stakeholders' security requirements, PERUN offers different throughput/latency performances for ML computations. For stakeholders willing strong integrity and confidentiality guarantees, PERUN executes ML computations only inside TEEs enclaves, *i.e.*, input and output data, code, and models never leave the enclave. For stakeholders accepting a larger trusted computing base in exchange for better performance, PERUN enables trusted computing techniques [73] to protect ML computations while executing them on hardware accelerators, *e.g.*, GPU. Specifically, it uses IMA, which is an integrity enforcement mechanism that prevents adversaries from running arbitrary software on the operating system, *i.e.*, software that allows reading data residing in the main memory or being transferred to or processed by the GPU. The security policy manager verifies that such a mechanism is en-

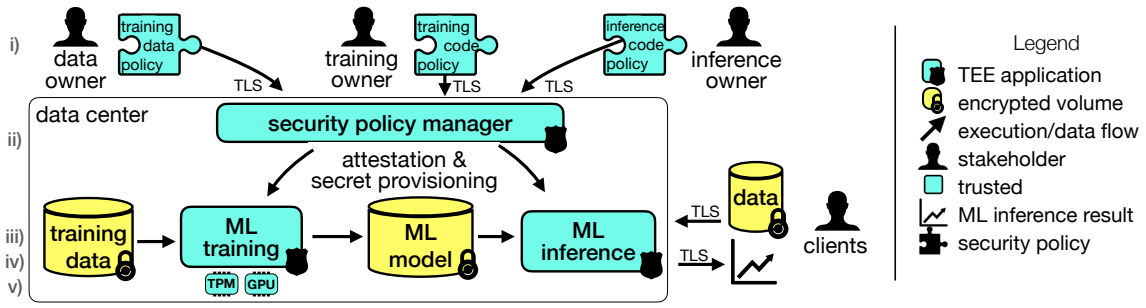


Figure 5.2. PERUN framework supports multi-stakeholder ML computation. Stakeholders trust the security policy manager. Inside security policies, they define which stakeholder's application can access a cryptographic key allowing decryption of confidential code or data. TEE protects code, data, and cryptographic keys.

abled by querying a secure element compatible with the trusted platform module (TPM) [90] attached to the remote computer.

5.4.2 Keys Sharing

Stakeholders use security policies to share encryption keys protecting their intellectual property. For example, the training data owner specifies in his security policy that he allows the ML computation of the training code owner to access his encryption key to decrypt the training data. The security policy manager plays a key role in the key sharing process. It generates an encryption key inside the TEE and securely distributes it to ML computations accordingly to the security policy. The training code owner cannot see the shared secret in the example above because it is transferred only to his application executing inside the TEE.

To provision ML computations with encryption keys, the security policy manager authenticates them using a remote attestation protocol offered by a TEE engine, *e.g.*, the SGX remote attestation protocol [133]. During the remote attestation, the TEE engine provides the security policy manager with a cryptographic measurement of the code executing on the remote platform. The cryptographic measurement – the output of the cryptographic hash function over the code loaded by the TEE engine to the memory – uniquely identifies the ML computation, allowing the security policy manager to authorize access to the encryption key based on the ML computation identity and stakeholder's security policies.

5.4.3 Security Policy and Trade-offs

PERUN relies on security policies as a means to define dependencies among stakeholders' computation and shared data.

Listing 5.1 shows an example of a policy. The policy has a unique name (Listing 5.1 line 1), typically combining a stakeholder's name and its intellectual property name. The name is used among stakeholders to reference *volumes* containing code, input, or output data. A volume is a collection of files encrypted with an encryption key managed by the security policy manager. Only the authorized ML computations have access to the key required to decrypt the volume and access the intellectual property. To prevent an adversary from changing the policy, the stakeholder embeds his public key inside the policy (Listing 5.1 line 21). The

security policy manager accepts only policies containing a valid signature issued with a corresponding stakeholder's private key.

The ML computation definition consists of a command required to execute the computation inside a container (Listing 5.1 line 2) and a cryptographic hash over the source code content implementing the ML computation (Listing 5.1 line 8). The security policy manager uses the hash to authenticate the ML computation before providing it with the encryption key.

The policy allows selecting trade-offs between security and performance. For example, a training code owner who wants to use the GPU to speed up the ML training computation might define conditions under which he trusts the operating system. In such a case, a stakeholder defines a certificate chain permitting to verify the authenticity of a secure element attached to the computer (Listing 5.1 line 10) and expected integrity measurements of the operating system (Listing 5.1 lines 16-19). The security policy manager only provisions the ML computations with the encryption key if the operating system integrity (kernel sources and configuration) are trusted by the stakeholder. Specifically, the operating system integrity measurements reflect what kernel code is running and whether it has enabled the required security mechanisms. Only then, the ML computations can access the confidential data and send it to the outside of the TEE, *e.g.*, GPU.

We discuss now and evaluate later (subsection 5.6.2) two security levels that are particularly important for the ML computation. The first one, the *high-assurance security level*, fits well the inference because it offers strong security guarantees provided by the TEE, allowing the inference model to execute in an untrusted data center controlled by an untrusted operator. It comes at performance limitation, which is acceptable for inference because, typically, inference operates on much smaller data than ML training and does not need access to hardware accelerators. The high-assurance security level offers confidentiality and integrity of code and data at rest and in runtime. The trusted computing base (TCB) is low; It includes only the inference model executing inside the TEE, the hardware providing the TEE functionality, and the key distribution process. The second security level, the *high integrity level*, fits well the ML training because it enables access to hardware accelerators required for intensive computation. It comes at the cost of a larger TCB compared to the high-assurance security level because the code providing access to the hardware accelerators, *i.e.*, an operating system, must be trusted. PERUN relies on the TPM to establish trust with load-time kernel integrity and on IMA to extend this trust to the operating system runtime integrity.

Listing 5.1: Security policy example

```
1  name: training_owner/training_code
2  command: python /app/training.py
3  volumes:
4      − path: /training_data
5      import: training_data_owner/training_data_service
6      − path: /inference_model
7      export: inference_owner/inference_service
8  integrity_hash: {"0a11...bb3f"}
9  operating_system:
10     certificate_chain: |−
11     −−−−−BEGIN CERTIFICATE−−−−−
12     # certificate chain allowing
13     # verification of the secure
```

```
14    # element manufacturer
15    −−−−−END CERTIFICATE−−−−−
16    integrity:
17    measure0: e0f1...4be6
18    measure1: ae44...3a6e
19    measure2: 3d45...796d
20  stakeholder: |−
21    −−−−−BEGIN PUBLIC KEY−−−−−
22    # the policy owner's public key
23    −−−−−END PUBLIC KEY−−−−−
```

5.4.4 Hardware ML Accelerators Support

Typically, ML computations (*e.g.*, deep neural networks training) are extremely intensive because they must process a large amount of input data. To decrease the computation time, popular ML frameworks, such as TensorFlow[11] [1], support hardware accelerators, such as GPUs or Google tensor processing units (TPUs). Unfortunately, existing hardware accelerators do not support confidential computing, thus not offering enough security guarantees for the multi-stakeholder ML computation. For example, an adversary who exploits an operating system misconfiguration [276] can launch arbitrary software to read data transferred to the GPU from any process executing in the operating system. Even if ML computations execute inside the TEE enclaves, an adversary controlling the operating system can read the data when it leaves the TEE, *i.e.*, it is transferred to the GPU or is processed by the GPU. Because of this, we design PERUN to support additional security mechanisms protecting access to the data (also code and ML models) while being processed out of the TEE. This also allows stakeholders to trade-off between security level and performance they want to achieve when performing ML computations.

Figure 5.3 shows how PERUN enables hardware accelerator support. ML computations transfer to the security policy manager a report describing the operating system's integrity state. The report is generated and cryptographically signed by a secure element, *e.g.*, a TPM chip, physically attached to the computer. The security policy manager authorizes the ML computation to use the encryption key only if the report states that the operating system is configured with the required security mechanisms. Precisely, the integrity enforcement mechanism, such as integrity measurement architecture (IMA) [225], controls that the operating system executes only software digitally signed by a stakeholder. Even if an adversary gains root access to the system, she cannot launch arbitrary software that allows her to sniff on the communication between the ML computation and the GPU, read the data from the main memory, or reconfigure the system to disable security mechanisms or load a malicious driver. This also allows PERUN to mitigate software-based micro-architectural and side-channel attacks [260, 40, 267, 82], which are vulnerabilities of TEEs.

To enable hardware accelerator support, a stakeholder specifies expected operating system integrity measurements inside the security policy (Listing 5.1 lines 9-19) and certificates allowing verification of the secure element identity. The operating system integrity measurements are cryptographic hashes over the operating system's kernel loaded to the memory during the boot process. A secure element collects such measurements during the boot pro-

[11] TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc.

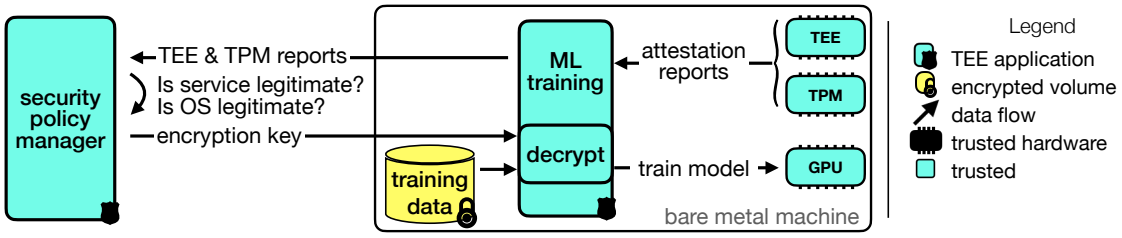


Figure 5.3. The high-level overview of PERUN supporting secure computation using hardware accelerator, *e.g.*, the GPU. PERUN performs both the SGX and TPM attestation before provisioning the ML code with cryptographic keys. The successful TPM attestation informs that the legitimate OS with enabled integrity-enforcement mechanisms controls access to the GPU.

cess and certifies them using a private key linked to a certificate issued by its manufacturer. The certificate and integrity measurements are enough for the security policy manager to verify that the IMA enforces the operating system integrity.

Although the hardware accelerator support comes at the cost of weaker security guarantees (additional hardware and software must be trusted when compared to a pure TEE-based approach), it greatly improves the ML training computation's performance (see subsection 5.6.2).

5.4.5 Zero Code Changes

PERUN framework requires zero code changes to run existing ML computations, thus providing a practical solution for legacy ML systems. To achieve it, PERUN adapts platforms supporting running legacy applications inside the TEE, such as SCONE [11] or GrapheneSGX [257]. These platforms allow executing unmodified code inside the TEE by recompiling the code using dedicated cross-compilers or running them with a modified interpreter executing in the TEE.

5.4.6 Policy Deployment and Updates

A stakeholder establishes a transport layer security (TLS) connection to the security policy manager to deploy a policy. During the TLS handshake, the stakeholder verifies the identity of the security policy manager. The security policy manager owns a private key and corresponding certificate signed by an entity trusted by a stakeholder. For example, such a certificate can be issued by a TEE provider who certifies that given software running inside a TEE and identified by a cryptographic hash is the security policy manager. Some TEE engines, such as SGX, offer such functionality preventing even a service administrator from seeing the private key [149]. For other TEEs, a certificate might be issued by a cloud provider operating the security policy manager as part of cloud offerings.

PERUN requires that the security policy manager authorizes changes to the deployed policy. Otherwise, an adversary might modify the stakeholder's policy allowing malicious code to access the encryption key. In the PERUN design, the stakeholder includes his public key inside the digitally signed security policy. Since then, the security policy manager accepts changes to the policy only if a new policy has a signature issued with the stakeholders' private key corresponding to the public key present in the existing policy. By having a public key embedded in the security policy, other stakeholders can verify that the policy is owned

by the stakeholder they cooperate with. The details of the policy security manager regarding key management, high availability, tolerance, and protection against rollback attacks are provided in [88].

5.5 Implementation

We implemented the PERUN prototype based on TensorFlow version 2.2.0 and the SCONE platform [11] because SCONE provides an ecosystem to run unmodified applications inside a TEE. We also rely on the existing key management system provided by the SCONE [258] and its predecessor [88] to distribute the configuration to applications. We rely on Intel SGX [45] as a TEE engine because it is widely used in practice.

Our prototype uses a TPM chip [90] to collect and report integrity measurements of the Linux kernel loaded to the memory during a trusted boot [235] provided by tboot [127] with Intel trusted execution technology (TXT) [87]. The Linux kernel is configured to enforce the integrity of software, dynamic libraries, and configuration files using Linux IMA [225], a Linux kernel's security subsystem. Using the TPM chip, PERUN verifies that the kernel is correctly configured and interrupts its execution when requirements are not met.

We use an nvidia GPU as an accelerator for ML computation. The ML services are implemented in Python using TensorFlow framework, which supports delegating ML computation to the GPU.

5.5.1 Running ML Computations Inside Intel SGX

To run unmodified ML computations inside the SGX enclaves, we use the SCONE cross-compiler and SCONE-enabled Python interpreters provided by SCONE as Docker images. They allow us to build binaries that execute inside the SGX enclave or run Python code inside SGX without any source code changes.

The SCONE wraps an application in a dynamically linked loader program (*SCONE loader*) and links it with a modified C-library (*SCONE runtime*) based on the musl libc [190]. On the ML computation startup, the SCONE loader requests SGX to create an isolated execution environment (enclave), moves the ML computation code inside the enclave, and starts. The SCONE runtime, which executes inside the enclave along with the ML computations, provides a sanitized interface to the operating system for transparent encryption and decryption of data entering and leaving the enclave. Also, the SCONE runtime provides the ML computations with its configuration using configuration and attestation service (CAS) [258].

5.5.2 Sharing the Encryption Key

We implement the security policy manager in the PERUN architecture using the CAS, to generate, distribute, and share encryption keys between security policies. We decided to use the CAS because it integrates well with SCONE-enabled applications and implements the SGX attestation protocol [133]. Other key management systems supporting the SGX attestation protocol might be used [37, 159] but require additional work to integrate them into SCONE.

We create a separate CAS policy for each stakeholder. The policy contains an identity of the stakeholder's intellectual property (data, code, and models) and its access control and configuration. It is uploaded to CAS via mutual TLS authentication using a stakeholder-specific private key corresponding to the public key defined inside the policy. This fulfills the

PERUN requirement of protecting unauthorized stakeholders from modifying policies. The intellectual property identity is defined using a unique per application cryptographic hash calculated by the SGX engine over the application's pages and their access rights. The SCONE provides this value during the application build process. The CAS allows for the specification of the encryption key as a program argument, environmental variable, or indirectly as a key related to an encrypted volume. Importantly, the CAS allows defining which policies have access to the key. Thus, with the proper policy configuration, stakeholders share keys among enclaves as required in the PERUN architecture.

Our prototype uses the CAS encrypted volume functionality, for which the SCONE runtime fetches from the CAS the ML computation configuration containing the encryption key. Specifically, following the SGX attestation protocol, the SCONE runtime sends to CAS the SGX attestation report in which the SGX hardware certifies the ML computation identity. The CAS then verifies that the report was issued by genuine SGX hardware and the ML computation is legitimate. Only afterward, it sends to the SCONE runtime the encryption key. The SCONE runtime transparently encrypts and decrypts data written and read by the ML computation from and to the volume. The ML computations, *i.e.*, training and inference authorized by stakeholders via policies, can access the same encryption key, thus gaining access to a shared volume.

5.5.3 Enabling GPU Support with Integrity Enforcement

Our prototype implementation supports delegating ML computations to the GPU under the condition that the integrity-enforced operating system handles the communication between the enclave and the GPU. The integrity enforcement mechanism prevents intercepting confidential data that leaves the enclave because it limits the operating system functionality to a subset of programs essential to load the ML computation and the GPU driver. Thus, a malicious program cannot run alongside the ML computations on the same computing resources. We use trusted boot and TPM to verify it, *i.e.*, that the remote computer runs a legitimate Linux kernel with enabled integrity enforcement that limits software running on the computer to the required operating system services, the GPU driver, and ML computations.

Trusted Boot

Trusted computing techniques (TCTs) define a set of technologies that measure, report, and enforce kernel integrity. Specifically, during the computer boot, we rely on a trusted bootloader [127], which uses a hardware CPU extension [87] to measure and securely load the Linux kernel to an isolated execution environment [235]. (The TXT session ends with the execution of tboot.) The trusted bootloader measures the kernel integrity (a cryptographic hash over the kernel sources) and sends the TPM chip measurements.

Integrity Enforcement

IMA is a kernel mechanism that authenticates files before allowing them to be loaded to the memory. Figure 5.4 shows how the IMA works. A process executing in userspace requests the kernel to execute a new application, load a dynamic library, or read a configuration file. IMA calculates the cryptographic hash over the file's content, reads the file's signature from the file's extended attribute, and verifies the signature using a public key stored in the kernel's

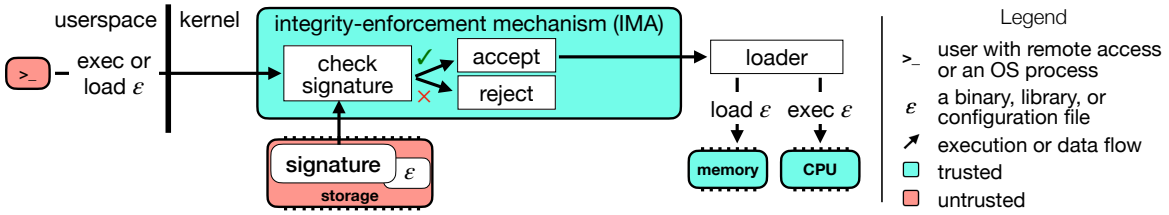


Figure 5.4. The kernel integrity-enforcement system authenticates a file by checking its digital signature before loading it to the memory.

ima keyring. If the signature is correct, IMA extends the hash (load-time integrity of the file) to a dedicated PCR and allows the kernel to continue loading the file.

Trusted Boot Service

Because SCONE is proprietary software, we could not modify the SCONE runtime to provide the CAS with the TPM report. Instead, we implemented this functionality in a *trusted boot service* that uses the TPM to verify that the ML computations execute in the integrity-enforced operating system. The trusted boot service relies on the CHORS's agent implementation introduced in §3.5.1.

The CAS performs the SGX attestation of the trusted boot service and provisions it with the TPM certificate as well as a list of the kernel integrity measurements. The trusted boot service reads the integrity measurements stored in PCRs using the TPM attestation protocol. The TPM genuineness is ensured by verifying the TPM certificate using a certificate chain provided by the CAS. The Linux kernel integrity is verified by comparing the integrity measurements certified by the TPM with the measurements read from the CAS.

We implemented the trusted boot service as an additional stage in the ML data processing. It enables other ML computations to access the confidential data only if the operating system state conforms to the stakeholder's security policy. It copies the confidential data from an encrypted volume of one ML computation to a volume accessible to another ML computation after verifying the kernel integrity using the TPM. Our implementation is complementary with Linux unified key setup (LUKS) [28]. LUKS allows the kernel to decrypt the file system only if the kernel integrity has not changed. This prevents accessing the trusted boot service's volume after modifying the kernel configuration, *i.e.*, disabling the integrity-enforcement mechanism.

5.6 Evaluation

Testbed. Experiments were executed on an ASUS Z170-A mainboard equipped with an Intel Core i7-6700K CPU supporting SGXv1, Nvidia GeForce RTX 2080 Super, 64 GiB of RAM, Samsung SSD 860 EVO 2 TB hard drive, Infineon OPTIGA™SLB 9665 TPM 2.0, a 10 Gb Ethernet network interface card connected to a 20 Gb/s switched network. Hyper-threading is enabled. The enclave page cache (EPC) is configured to reserve 128 MB of RAM. CPUs are on the microcode patch level 0xe2. We run Ubuntu 20.04 with Linux kernel 5.4.0-65-generic. Linux IMA is enabled. The hashes of all operating system files are digitally signed using a 1024-bit RSA asymmetric key. The signatures are stored inside files' extended attributes,

and the certificate signed by the kernel's build signing key is loaded to the kernel's keyring during initrd execution.

Datasets. We use two datasets: (i) the classical CIFAR-10 image dataset [157], and (ii) the real-world medical dataset [236].

5.6.1 Attestation Latency

We run an experiment to measure the overhead of verifying the operating system integrity using the TPM. Precisely, we measure how much time it takes an application implementing the trusted boot service to receive configuration from the security policy manager, read the TPM, and verify the operating system integrity measurements.

The security policy manager executes on a different machine located in the same data center. It performs the SGX attestation before delivering a configuration containing two encryption keys – a typical setup for ML computations – and measurements required to verify the operating system integrity. The security policy manager and the trusted boot service execute inside SCONE-protected Docker containers.

Table 5.1. End-to-end latency of verifying software authenticity and integrity using SGX and TPM attestation. Mean latencies are calculated as 10% trimmed mean from ten independent runs. *sd* stands for standard deviation.

	Execution time
Application in a container	1573 ms (sd=16 ms)
+ SGX attestation	1691 ms (sd=37 ms)
+ SGX and TPM attestation	2410 ms (sd=33 ms)

Table 5.1 shows that launching the application inside a SCONE-protected container takes 1573 ms. Running the same application that additionally receives the configuration from the security policy manager incurs 118 ms overhead. Additional 719 ms are required to read the TPM quote, verify the TPM integrity and authenticity, and compare the read integrity measurements with expected values provided by the security policy manager. As we show next, 2.5 sec overhead required to perform SGX and TPM attestation is negligible considering the ML training execution time.

5.6.2 Security and Performance Trade-off

To demonstrate the advantage of PERUN in allowing users to select the trade-off between security and performance, we compare the performance of different security levels provided by PERUN and the pure SGX based system called SecureTF [167]. We run the model training using the following setups: (i) only CPU (*Native*); (ii) GPU (*Native GPU*); (iii) PERUN, IMA enabled (*PERUN+IMA*); (iv) PERUN, IMA and SGX enabled (*PERUN+IMA+SGX*); (v) PERUN with GPU, IMA enabled (*PERUN+IMA+GPU*).

The *Native* and *Native GPU* levels represent scenarios where no security guarantees are provided. *PERUN+IMA* and *PERUN+IMA+GPU* represent the high integrity level (subsection 5.4.3) in which ML training can execute directly on the CPU or GPU (high performance) while require to extend trust to the operating system (large TCB). Finally, *PERUN+IMA+SGX* represents the high-assurance security level where all computations are performed inside the TEE (limited performance) but requires a minimal amount of trust in the remote execution environment (low TCB). In all setups, the trusted boot service executes inside the enclave.

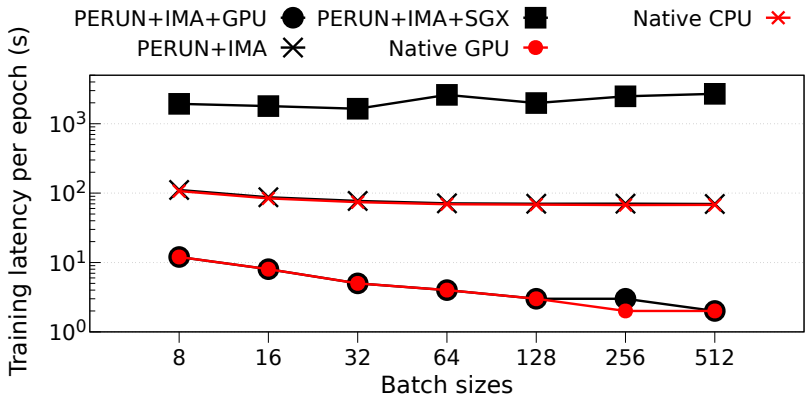


Figure 5.5. The CIFAR-10 training latency comparison among different security levels offered by PERUN. Mean latencies are calculated from five independent runs.

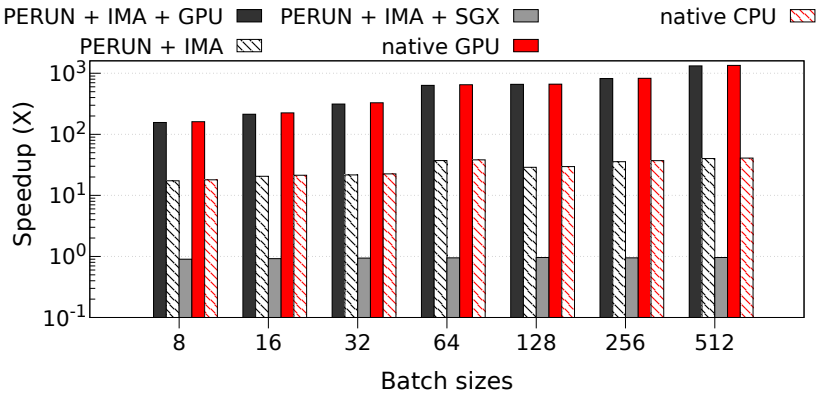


Figure 5.6. The CIFAR-10 training speedup of evaluated systems in comparison to PERUN with the highest security level (PERUN+IMA+SGX).

CIFAR-10 Dataset

We perform training using the CIFAR-10 dataset, a convolutional neural network containing four *conv* layers followed by two fully connected layers. We use *BatchNorm* after each conv layer. We apply the *ADAM* optimization algorithm [146] with the learning rate set to 0.001.

Figure 5.5 shows the training latency, and Figure 5.6 shows the PERUN speedup depending on setups and batch sizes. At the high-assurance security level (*PERUN+IMA+SGX*), PERUN achieves almost the same performance as the pure SGX-based system, secureTF. This is because the training data is processed only inside the enclave, and SGX performs compute-intensive paging caused by the limited EPC size (128 MB) that cannot accommodate the training computation data (8 GB). *PERUN+IMA+GPU* and *PERUN+IMA* achieve 1321× and 40× speedup when relying just on the high integrity level compared to secureTF (batch size of 512). With these setups, the PERUN performance is similar to native systems (∼ 0.96× of native latency) because the integrity protection mechanism performs integrity checks only when it loads files to the memory for the first time, leading to almost native execution afterward.

Real-world Medical Dataset

Next, we evaluate PERUN using a large-scale real-world medical dataset [236]. The dataset contains a wide range of medical images, including images of cancer and tumor treatment regimens for various parts of the human body, *e.g.*, brain, colon, prostate, liver, and lung. It was created via CT or MRI scans by universities and research centers from all around the world. We perform training over the brain tumor images dataset (6.1 GB) using the 2-D U-Net [221] TensorFlow architecture from Intel AI [5]. It makes use of the *ADAM* optimizer that includes 7 760 385 parameters with 32 feature maps. We set the learning rate to 0.001 and the batch size to 32.

Table 5.2. The training latency comparison among different security levels of PERUN, secureTF, and native. The results were obtained from a single run.

System	Latency per epoch	Speedup
Native CPU	5 h 26 min 14 sec	47×
Native GPU	9 min 54 sec	1561×
PERUN+IMA	5 h 26 min 17 sec	47×
PERUN+IMA+SGX	257 h 27 min 49 sec	∼ 1×
PERUN+IMA+GPU	9 min 55 sec	1560×
secureTF	257 h 43 min 53 sec	(baseline)

Table 5.2 shows that at the high-assurance security level (the data is processed entirely inside the enclave), *PERUN+IMA+SGX* achieves the same performance as the referenced SGX-based system. However, when relying just on the high integrity level to protect the data, *PERUN+IMA+GPU* and *PERUN+IMA* achieve a speedup of 1559× and 47× compared to secureTF, respectively. We maintain the accuracy of 0.9875 in all experiments (dice coef: 0.5503, soft dice coef: 0.5503).

5.7 Related Work

5.7.1 Secure Multi-party Computation

Although cryptographic schemes, such as secure multi-party computation (MPC) and fully homomorphic encryption, are promising to secure multi-stakeholder ML computation, they have limited application in practice [263, 199]. They introduce high-performance overhead [199, 187, 160, 188, 137], which is a limiting factor for computing-intensive ML, and require to heavily modify existing ML code. Furthermore, they do not support all ML algorithms, such as, deep neural networks. Some of them also require additional assumptions, like MPC protocol requiring a subset of honest stakeholders. Unlike PERUN, most of them lack support for training computation. Instead, PERUN requires zero-code changes to the ML applications, supports multi-stakeholder ML training, and offers good performance at the cost of much larger TCB than the pure cryptographic solutions.

5.7.2 Secure ML using TEEs

Many works leverage TEE to support secure ML [94, 107, 199]. Chiron [107] uses SGX for privacy-preserving ML services, but it is only a single-threaded system. Also, it needs to add

an interpreter and model compiler into the enclave. This incurs high runtime overhead due to the limited EPC size. The work from Ohrimenko et al. [199] also relies on SGX for secure ML computations. However, it does not allow using hardware accelerators and supports only a limited number of operators — not enough for complex ML computations. In contrast to these systems, PERUN supports legacy ML applications without changing their source code. SecureTF [167] is the most relevant work for PERUN because it also uses SCONE. It supports inference and training computation, as well as distributed settings. However, it is not clear how secureTF can be extended to support secure multi-stakeholders ML computation. Also, secureTF does not support hardware accelerators, making it less practical for training computation. Other works [253, 192, 13] use SGX and untrusted GPUs for secure ML computations. They split ML computations into trusted parts running in the enclave and untrusted parts running in the GPU. However, they require changing the existing code and do not support multi-stakeholder settings.

5.7.3 Trusted GPUs

Although trusted computation on GPUs is not commercially available, there is ongoing research. HIX [131] enables memory-mapped I/O access from applications running in SGX by extending an SGX-like design with duplicate versions of the enclave memory protection hardware. Graviton [264] proposes hardware extensions to provide TEE inside the GPU directly. Graviton requires modifying the GPU hardware to disable direct access to the critical GPU interfaces, *e.g.*, page table and communication channels from the GPU driver. Telekine [106] restricts access to GPU page tables without trusting the kernel driver, and it secures communication with the GPU using cryptographic schemes. The main limitation of these solutions is that they require hardware modification of the GPU design, so they cannot protect existing ML computations, and they also do not support multi-stakeholder ML computations.

5.8 Summary

PERUN allows multiple stakeholders to perform ML without revealing their intellectual property. It provides strong confidentiality and integrity guarantees at the performance of existing TEE-based systems. With the help of trusted computing, PERUN permits utilizing hardware accelerators, reaching native hardware-accelerated systems' performance at the cost of a larger trusted computing base. When training an ML model using real-world datasets, PERUN achieves 0.96× of native performance execution on the GPU and a speedup of up to 1560× compared to the state-of-the-art SGX-based system.

6 A Practical Approach For Updating an Integrity-enforced Operating System

Techniques presented in chapter 3, chapter 4, and chapter 5 rely on trusted computing techniques (TCTs) to measure, record, enforce, and report the integrity of an operating system. While promising at first glance, these systems, as well as any other system leveraging TCTs, suffer from limitations when deployed in production. Specifically, they do not support operating system updates because the security patches, which might be released frequently and installed automatically, break the operating system's integrity. We refer to integrity as a security property describing that a computer runs only expected software in the expected configuration.

To illustrate the problem of installing software updates, we first describe the concept of integrity verification provided by TCTs technologies. Verifiers (*e.g.*, monitoring systems [125, 128, 111] or virtual private network access points [240]) use hardware and software technologies [87, 225, 121], which implement trusted computing techniques [235, 90, 89], to identify compromised (executing not allowed software) or misconfigured (having invalid configuration) systems. In more detail, verifiers read from a remote computer a list of cryptographic hashes, in the form of a *measurement report*, calculated over every file loaded to the computer memory since the computer boot. Verifiers detect integrity violations by comparing hashes to a whitelist, which is a list that contains hashes of approved software and configuration. Unfortunately, verifiers cannot distinguish whether software integrity changed due to malicious behavior or a legitimate software update (see Figure 6.1).

Berger et al. 2015 [22] proposed to include in the measurement report digital signatures, which certify the integrity hashes of trusted software. The approach simplifies the verification process because verifiers require only a single certificate to check the signatures instead of a whitelist of all possible cryptographic hashes. Consequently, it opened an opportunity to support the operating system's updates because updates could incorporate digital signatures to vouch for the integrity of files changed during the update. This approach has, however, two limitations, which we address in this chaper. First, it requires changes to the existing procedures of creating packages for every operating system distribution because each operating system distribution would have to issue and insert digital signatures of files inside their packages [23]. Second, software packages contain not only files that are extracted to

the filesystem but also configuration scripts that might alter the operating system's configuration, thus breaking the integrity.

Instead of modifying the well-established process of package generation, which requires approval from the entire open-source community, an alternative approach consists of creating a standalone repository with modified packages containing digital signatures [23]. The approach requires a trusted organization that owns a signing key and re-creates packages after injecting digital signatures. Such an organization must put additional efforts to protect the signing key and must have a good reputation to convince users to trust it. We argue that it might be difficult to achieve, considering incidents from the past, when signing keys of major Linux distribution were leaked affecting millions of users [72, 214].

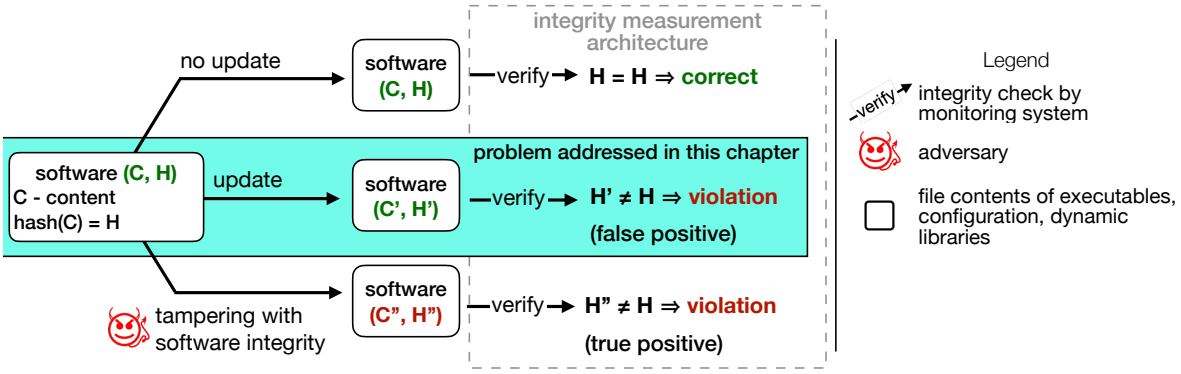


Figure 6.1. Problem of installing software updates in an integrity-enforced operating system. Software updates change software integrity measurement, which is reported by the monitoring systems as integrity violation. The main question addressed in this chapter: How to distinguish between software manipulated by an adversary and correctly updated software?

Another problem is that an adversary controlling a repository can provide the operating system with outdated packages containing known vulnerabilities (replay attack), or even prevent the operating system from seeing the update (freeze attack) [32, 33]. The secure choice is to rely only on the *original repository*, which is a repository managed by a trusted organization, such as an official software repository of the operating system distribution. But, this approach does not tolerate the original repository failure, thus the operating system must also accept *mirrors*. Mirrors store a copy of the original repository, and, in the case of open-source distributions, are hosted voluntarily. As reported by previous studies [32], it is not difficult to create a custom mirror that becomes accepted as an official mirror. Therefore, we must tolerate that some of the available mirrors are controlled by an adversary, exposing operating systems to threats mentioned above. For example, it happened that a compromised mirror of a popular repository distributed a vulnerable version of the software, allowing an adversary to remotely access the system [186].

6.1 Contribution

We present ROD, an intermediate layer between the operating system and the software repository that provides *sanitized* software packages. The installation of sanitized packages causes deterministic changes to the operating system configuration and filesystem. Because such changes are verifiable by monitoring systems, ROD eliminates the risk of false-positives. According to our measures, sanitization enables 99.76% of packages available in the Alpine

main and community repositories to be safely installed in integrity-enforced operating systems.

ROD requires zero code changes to both monitoring systems as well as operating systems. Due to the shared nature of the software repositories, we designed ROD as a service that can be hosted on third-party resources, *i.e.*, in the cloud. ROD exploits trusted execution environment (TEE), *i.e.*, Intel software guard extensions (SGX) [45, 185], to protect the signing keys and ROD integrity. Our evaluation shows that running ROD inside SGX is practical; SGX induces in average 1.18× performance overhead during sanitization, up to 1.96× for packages exceeding available SGX memory. Note that the sanitization is performed in batch mode and hence, the slowdown has no practical impact.

Last but not least, ROD accepts security policies, which reflect organizational-specific security requirements. Specifically, each organization defines a list of mirrors, which ROD uses to establish quorum on the correct version of a software package, thus tolerating mirrors compromised by an adversary. We show that ROD requires up to 2.2 seconds to establish a quorum from official Alpine mirrors distributed over three continents.

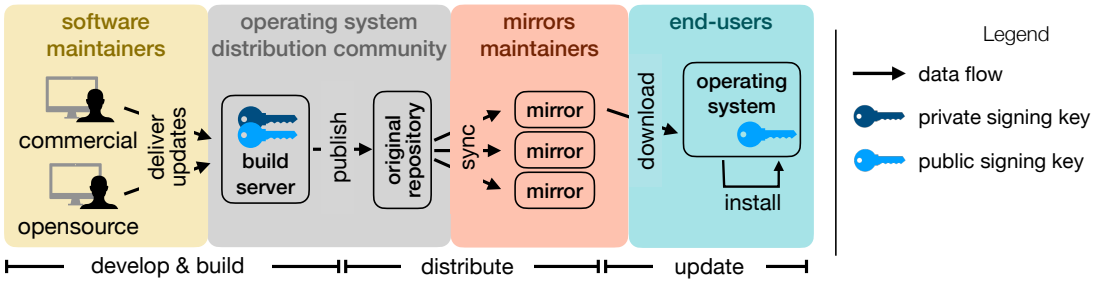


Figure 6.2. Overview of software update process. Colors indicate different administrative domains and are consistent across all figures in this chapter.

To summarize, in this chapter, we make the following contributions: (i) We propose a practical solution to support operating system updates in integrity-enforced systems, allowing for software packages to be safely installed in integrity-enforced operating systems (§6.5.2), transparent support of the existing software update processes and infrastructure (§6.5.3), tolerance of a minority of mirrors exhibiting Byzantine behavior (§6.5.5). (ii) We realize the above-mentioned design by developing ROD— a secure proxy framework for supporting software updates in integrity-enforced operating systems (§6.6). (iii) We have evaluated ROD using a series of micro-benchmarks, and a real-world use case — Alpine Linux package updates (§6.7).

6.2 Background

To better understand the decisions taken in designing ROD, we start by providing background information on software update processes and about existing technologies used to collect, report, and verify system integrity.

6.2.1 Operating System Updates

Figure 6.2 shows a high-level overview of an operating system update process: releasing, exposing, and installing new software versions. The process begins when software maintain-

ers create a new software release that contains bug fixes or new features. The operating system distribution community uses the source code of the new software release to create a software package. A software package is an archive containing software-specific files and meta-information required by the operating system to install and manage the package. Packages are stored in a repository, from which end-users download them. A repository also stores a *metadata index* that contains a digitally signed list of all packages. In this chaper, we refer to a software repository controlled by an operating system distribution community as an *original repository*. The original repository is a root of trust for software updates. The metadata file downloaded from the original repository provides information about the most recent versions of software available in the repository. As such, it can be used to verify that the operating system is up-to-date.

Repository mirrors contain a copy of the original repository. They are used to distribute the load and to decrease the latency of downloading packages. The community has limited control over the mirrors, which are typically supported by volunteer organizations. Importantly, mirrors do not have access to the signing key. End-users verify that the metadata file and packages downloaded from mirrors originate from the original repository by verifying digital signatures using a public portion of the signing key provided by the operating system distribution community.

6.2.2 Package Managers

Operating systems use *package managers* to simplify installation, update, and removal of software. The majority of distributions ship with package managers that use pre-built packages (*e.g.*, .rpm, .deb [176], .apk [175]), but some build software directly from sources [177, 10]. In this chaper, we focus only on the pre-built packages, which we refer to further as *packages*.

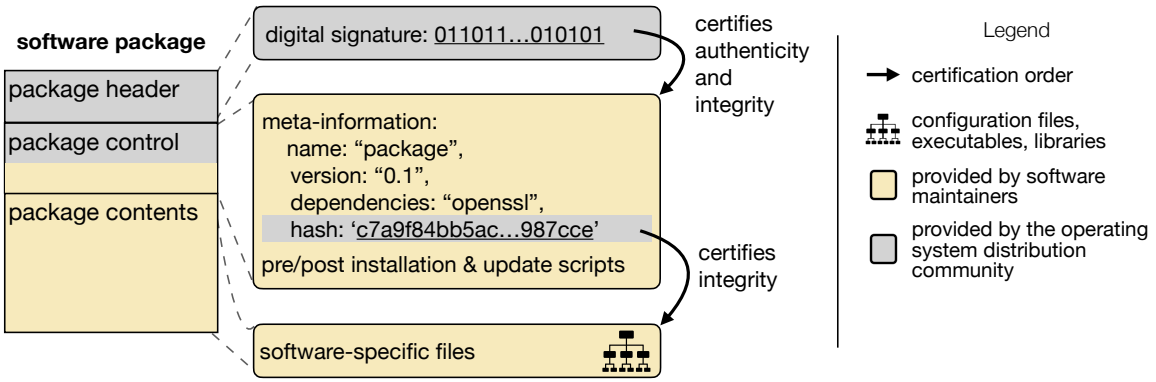


Figure 6.3. The internal structure of a software package, *i.e.*, Alpine APK package format. The package authenticity and integrity can be verified by using the digital signature and the content hash. The digital signature is stored inside the header, and is issued over the package control. The hash of the package contents is stored inside the meta-attributes of the package control.

A package is an archive containing software-specific files, installation scripts, meta-information (such as dependency on other packages), and digital signatures. Figure 6.3 shows an example of a package in the Alpine Linux .apk format. The package header stores a digital signature issued by a developer with an offline signing key, a private key stored off the repository. The digital signature permits verifying the authenticity and the integrity of the package control, which contains installation scripts and meta-information describing package dependencies,

software version, and a cryptographic hash of package contents. The hash permits verifying the integrity of executables, dynamic libraries, and configuration files stored inside the package.

To install the package, the package manager first downloads it from the repository, or from middlemen such as a content delivery network (CDN) or mirrors. After that, it verifies that a trusted entity created the package. Finally, it runs installation scripts and extracts software-specific files to the file system.

6.3 Threat Model

We assume an adversary whose goal is to install vulnerable software on a remote computer by exploiting the software update mechanism. A remote computer is configured to install updates from ROD, which itself relies on the original repository and official mirrors. An adversary has root access to the machine running ROD and to the minority of machines hosting mirrors. In more detail, she controls up to f mirrors out of a total of $2f + 1$ mirrors available to ROD. The adversary has access to all outdated packages that contain vulnerabilities, including outdated signed metadata files. By having root access to machines hosting ROD and mirrors, she can prevent network connection to the original repository and arbitrary mirrors.

We assume that the operating system distribution community, software maintainers, their internal processes (*i.e.*, software development, packages build), and infrastructure are trusted. In particular, packages are built using legitimate compilers; signing keys are well protected; the original repository provides the most recent software versions. We do not consider attacks resulting from the incorrect design of package formats and metadata, *i.e.*, the endless data attack and the extraneous dependencies attack [32]. The assumption is practical because main repositories hosted by the popular Linux distributions (*i.e.*, Debian, Ubuntu, RedHat, Alpine) and their corresponding package managers mitigate the attacks by digitally signing the metadata, which also includes packages file sizes and integrity hashes.

The TEEs are vulnerable to side-channel attacks [151, 260]. We exclude them from the threat model, assuming they can be addressed using dedicated tools [34, 200, 201], by updating microcode [126], or by excluding a particular type of hardware during the remote attestation protocol [133].

6.4 Problem Statement

We now introduce the main challenges and problems that shaped the ROD design.

Problem 1: How to modify the package so that the changes made to the operating system configuration and filesystem are verifiable by the monitoring system?

The monitoring systems regularly verify that remote computers run only expected software in the expected configuration. Machines that fail the attestation might be restarted or reinstalled to bring the system back into the correct state. Also, there exist mechanisms to enforce operating system integrity locally. Such mechanisms are built into the kernel (*e.g.*, IMA-appraisal [101]), allowing the kernel to authorize each file before loading it to the memory. They make the integrity attestation more robust, preventing accidental or malicious changes to the filesystem.

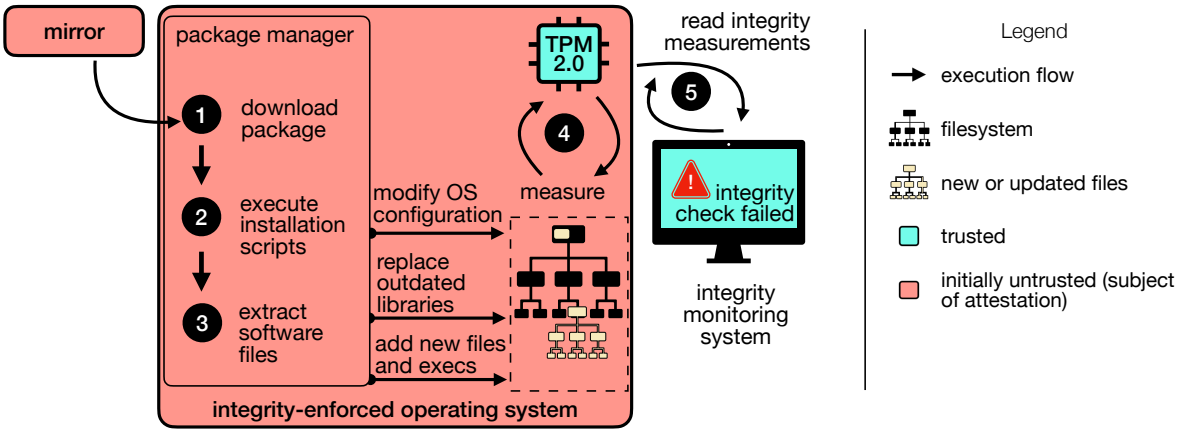


Figure 6.4. Example of the package installation that changes the operating system configuration and filesystem. Monitoring systems consider such a system compromised because the new operating system configuration might, for example, allow an adversary to get remote access to the computer or remotely exploit vulnerabilities in the replaced dynamic libraries.

The main problem of applying trusted computing in production systems is, however, that software updates cannot be safely installed because they modify the operating system configuration and change files in a way unknown to monitoring systems. Figure 6.4 shows why the package installation might move the operating system into an untrusted state. After the package is downloaded (❶), the package manager executes software-specific installation scripts that modify the operating system configuration (❷). Moreover, the package manager extracts software-specific files (❸), which contents are not known to verifiers. The integrity of the operating system configuration files and software-specific files is measured by trusted computing components (❹). Eventually, a monitoring system uses remote attestation to read the measurements (❺), thus detecting the operating system integrity change. The operating system is considered compromised.

A strawman approach consists of providing the monitoring system with a list of valid measurements before installing a new package. In practice, constructing such a list a priori is a difficult problem because of the complex nature of software dependencies, the operating system configuration depending on the order in which software has been installed, and unpredictable schedules of security updates.

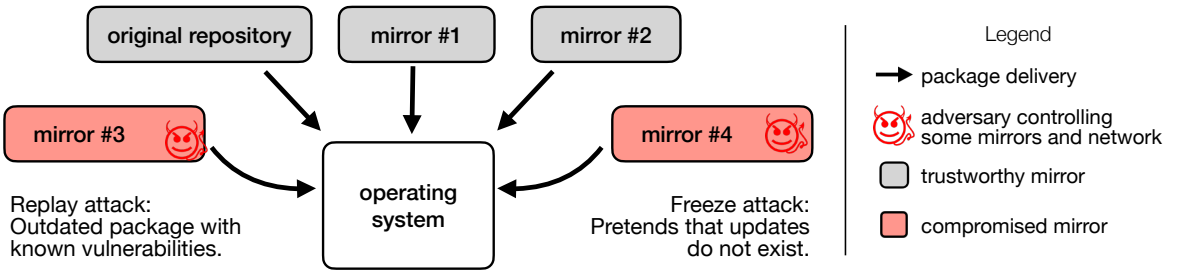


Figure 6.5. Mirrors controlled by an adversary can provide outdated packages with known vulnerabilities (replay attack) or completely hide the presence of software updates (freeze attack). An adversary might prevent access to the original repository (the root of trust) forcing the operating system to rely on mirrors.

Problem 2: How to modify packages without changing the well-established package creation requiring community approval?

Previous studies proposed changing the package creation process operated by different Linux communities to include digital signatures that vouch for individual file integrity [22]. Although different approaches have been proposed [239, 181], they have not gained enough community approval and have not been merged into upstream repositories. Therefore, a practical solution should not require changes to the existing package creation processes, thus be transparent to the existing update infrastructure and processes.

Problem 3: How to protect the signing key and to guarantee the correct generation of signatures in the presence of a powerful adversary with administrative access to ROD?

If we assume that we know how to modify the package (Problem 1), the operating system would reject the modified package because its digital signature would not match the package contents. This is expected behavior because it prevents operating systems from installing packages tampered by an adversary. Therefore, a new package content must be certified again. However, without community support, it is impossible to issue the signature because the community would restrict access to the signing key (Problem 2).

An alternative approach is to let ROD generate a custom signing key, so it uses it to sign all modified packages. However, an adversary with access to the machine on which the signing keys are used might extract the signing key by simply reading the process memory using administrative rights or by exploiting memory corruption techniques [171]. Consequently, the adversary might sign arbitrary packages compromising all operating systems that trust the signing key.

Problem 4: How to ensure access to the most up-to-date packages despite having no connection to the main software repository?

Software repositories are maintained by the operating system distributions and provide public access to packages and updates. We refer to such repositories as *original repositories* because new versions of packages and software updates are published directly there. Although the secure choice would be to always rely on the original repository controlled by a trusted organization, such a decision would introduce a single point of failure. For this reason, original repositories propagate software updates to mirrors, which expose them to a wide range of end client machines.

As reported by previous studies, an adversary controlling the mirror can serve outdated, vulnerable packages, decreasing the security of operating systems relying on that mirror [32, 33]. Figure 6.5 shows that an adversary might prevent the operating system from accessing the original repository, and forcing the operating system to use mirrors under her control.

6.5 Approach: Trusted Software Repository

Our objective is to provide an architecture that:

- provides software updates which can be safely installed in an integrity-enforced operating system,

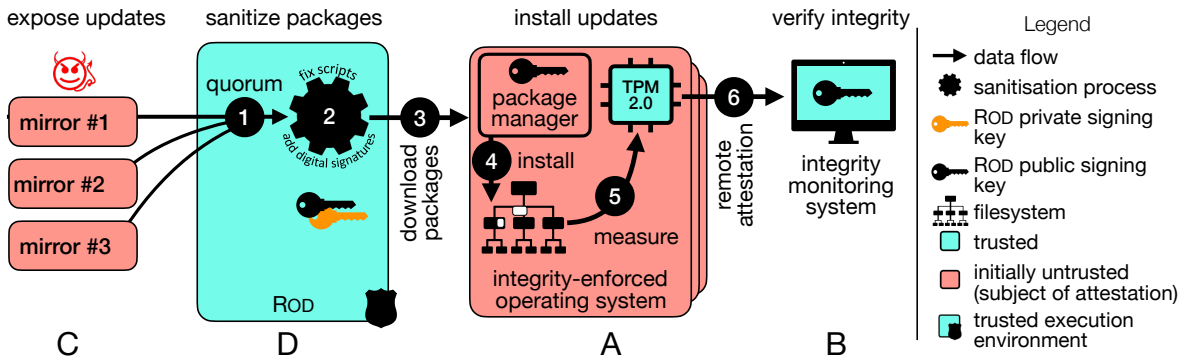


Figure 6.6. High-level overview of trusted software repository (ROD) architecture. ROD is a proxy that modifies packages in a way they are safe to be installed in the integrity-enforced operating systems. ROD, TPM, and the integrity monitoring system are trusted.

- requires no changes to the process of how communities create and distribute software packages,
- tolerates threats defined in §6.3.

6.5.1 Design

Figure 6.6 shows a high-level overview of the ROD design. It consists of four components: *(A)* an integrity-enforced operating system measured by trusted computing components, *(B)* a monitoring system which remotely verifies operating system integrity, *(C)* mirrors, copies of the original repository, containing operating system-dependent software packages, *(D)* ROD, an intermediate layer that provides the operating system with access to software packages that are safe to install in an integrity-enforced operating system.

Now, we present how ROD integrates with the software update process. First, ROD fetches the most up-to-date packages from mirrors (❶) and modifies them in a way they are safe to install (❷). Next, the package manager queries ROD to collect information about the latest versions of packages. After selecting packages to update, it downloads them from ROD (❸). Then, the package manager installs them (❹), causing partial update of the existing operating system configuration, replacement of existing files (*e.g.*, dynamic libraries), and extraction of new files into the filesystem. Trusted computing components regularly measure these changes, and the corresponding integrity measurements are stored inside a trusted platform module (TPM) chip (❺). The monitoring system collects the attestation report (❻), which next to integrity measurements, contains the corresponding digital signatures. After verifying the digital signatures and the integrity measurements, the monitoring system accepts a new state of the updated operating system.

6.5.2 Solution to Problem 1: Sanitization

To enable support for software updates, we must solve two problems. First, convince a monitoring system that the integrity measurements of files extracted from the software package to the operating system are valid. Second, make sure that the execution of a software package installation script does not cause the transition of the operating system into an untrusted state.

Table 6.1. Number of packages with and without custom configuration scripts in Alpine Linux main and community repositories. Some packages (Safe=✗) contain scripts that break the operating system's integrity.

Alpine repository			
Packages in			
Main	Community		
5665	5916	Total	Safe
5531	5772	Without scripts	✓
24	29	With safe scripts	✓
110	115	With unsafe scripts	✗

To address these problems, we introduce the concept of package *sanitization* (Figure 6.6 (❷)). The sanitization consists of verifying and modifying packages by: (i) changing installation scripts to ensure that their execution changes the operating system configuration in a deterministic way; (ii) predicting such configuration; (iii) including digital signatures of files delivered with the software package and the predicted operating system configuration.

Digital Signatures

Following the work of Berger et al. 2015, we propose that for each file stored inside a package, a corresponding digital signature certifying its integrity is also stored inside the package. The package manager would extract digital signatures to the filesystem, allowing the IMA to include digital signatures inside the attestation report. Consequently, the verifiers could recognize that the new integrity measurements are valid because they correspond to installation scripts and package-specific files.

Installation Scripts

Software packages might contain scripts that are executed with administrative rights during the package installation. Developers or package creators provide such scripts, and there are no limitations on what kind of operating system configuration changes scripts can do. Therefore it is possible that, due to a misconfiguration, a script reconfigures the operating system, allowing remote access to the machine. We designed ROD to modify packages in such a way the installation scripts change operating system configuration deterministically. The packages in which scripts cannot be sanitized are rejected from ROD, and thus not available for installation.

To design the script sanitization algorithm, we started by analyzing existing scripts wrapped inside packages available in the Alpine Linux repositories[12]. Table 6.1 shows that 97.6% of packages do not contain any scripts. 81% of the remaining packages contain scripts that alter the operating system configuration, breaking the system integrity.

We analyzed commands executed inside the scripts to understand how they interfere with the operating system configuration. Table 6.2 shows that 45 packages modify the filesystem structure (*i.e.*, copying, moving, or removing files, directories, and symbolic links, also changing their permissions). From the operating system integrity point of view, these actions are

[12] v3.11 of the Alpine Linux main [174] and community [173] repositories.

Table 6.2. Operations performed by installation scripts located in software packages in Alpine Linux repositories. Some operations (Safe=✗) break the operating system's integrity. The last column ("ROD ") indicates which operations are safe after the sanitization. *Filesystem changes*: add/remove/modify folders, symbolic links, and their permissions. *Empty scripts*: conditional checks, display information.

Operations executed in scripts				
Packages in				
Main	Community	Type	Safe	ROD
30	15	Filesystem changes	✓	✓
5	17	Empty scripts	✓	✓
17	19	Text processing	✓	✓
11	7	Configuration change	✗	✗
1	0	Empty file creation	✗	✓
97	104	User/Group creation	✗	✓
4	6	Shell activation	✗	✗

safe – they do not violate system integrity as defined by the IMA. Similarly, 36 packages execute text processing utilities (*e.g.*, parsing existing operating system configuration), which do not alter any existing file; thus, they are safe. However, 230 packages contain scripts modifying the operating system configuration, creating new users and groups, activating new shells, or creating empty files. These scripts are unsafe because they modify existing file contents in which integrity is certified using pre-generated signatures (as discussed in the previous section).

Script Sanitization

As we show next, the majority of the unsafe scripts provide a predictable output. Hence it is possible to predict the operating system configuration before installing the package.

The installation or update of 201 packages results in the creation of new users or groups. In the case of Linux-based operating systems, three files are affected, *i.e.*, /etc/passwd, /etc/-group, /etc/shadow. Interestingly, these files change in a deterministic way. Adding a new user or group results in adding a new well-defined line in at least one of these files. However, the order in which users and groups are created determines final file contents. In particular, different package installation order results in a different order in which users and groups are defined inside of each file.

Our solution consists of scanning the entire repository to learn about all possible users and groups that might be added by any software package. Then, we change each installation script in each package in a way the script creates all possible users and groups in the same predefined order. Consequently, any selection of packages and their order always results in the same operating system configuration – it contains all users and groups. Finally, ROD issues digital signatures over the predicted contents of the configuration files and modifies scripts to install the signatures in the target operating system. Monitoring systems accept the new operating system configuration because they read a measurement report containing the signatures, which vouch for the new configuration files contents.

Our ROD implementation detected and sanitized two packages that not only create a user but also set an empty password and shell. Installation of such packages might cause a security breach by allowing an adversary to remotely connect to the operating system using a

well-known username and password [194].

Unsupported Scripts

ROD does not support 28 packages (0.24%) out of all packages available in Alpine repositories. In particular, ROD does not support packages in which installation changes arbitrary configuration files. For example, a package *roundcubemail* is not supported because it generates an unpredictable configuration file containing a random session key. Although ROD could support it by generating the session key during the sanitization, such a solution would contradict the script functionality that provides a unique key per the operating system.

On the other hand, ROD intentionally does not support software packages providing different shells (*e.g.*, mksh, bash, tcsh). Their scripts modify the operating system configuration by activating a newly installed shell using *add-shell* command. Although ROD might use the same technique as with adding users and groups, we argue that the installation of a custom shell should not occur during an operating system update but should instead be part of the initial operating system configuration.

6.5.3 Solution to Problem 2: Proxy

We designed ROD as a proxy between package managers and software repositories provided by the community. This design decision permits ROD to act as a separate software repository that serves sanitized packages signed directly by ROD. From the community point of view, no changes are required to the existing software package creation processes, software package formats, or the implementation of package managers. Package managers recognize ROD as a standard repository mirror. Hence, it is enough to adjust the operating system configuration in a way the package manager uses only ROD as a mirror.

6.5.4 Solution to Problem 3: Shielded Execution

ROD requires a signing key to certify changes made to packages during the sanitization process. To protect the signing key from an adversary with root access to the machine, we propose to use TEE. In particular, we propose to leverage SGX, which is Intel's CPU extension providing confidentiality and integrity guarantees to applications running in environments in which the operating system, hypervisor, or basic input/output system (BIOS) might have been compromised. Other studies [88] demonstrated that applications running inside an enclave (a trusted execution environment provided by SGX) can generate, store, and use cryptographic keys that are only known to the specific application – not even a human being can read them. ROD's design relies on that concept. By running ROD inside an enclave, ROD generates a signing key that is used later to sign all modified software packages. The public portion of the signing key is exposed to both operating systems and monitoring systems that use it to verify that software packages were created by ROD.

Listing 6.1: Policy example of ROD

```
1 mirrors:
2   - hostname: https://alpinelinux/v3.10/
3     certificate_chain: |-
4       -----BEGIN CERTIFICATE-----
```

```
    (...)
    -----END CERTIFICATE-----
  - hostname: https://yandex.ru/alpine/v3.10/
    certificate_chain: |-
    -----BEGIN CERTIFICATE-----
    (...)
    -----END CERTIFICATE-----
  - hostname: https://ustc.edu.cn/alpine/v3.10/
    certificate_chain: |-
    -----BEGIN CERTIFICATE-----
    (...)
    -----END CERTIFICATE-----
signers_keys:
  - |- # e.g., alpine@alpinelinux.org-4a40.rsa.pub
    -----BEGIN PUBLIC KEY-----
    (...)
    -----END PUBLIC KEY-----
  - |- # e.g., alpine@alpinelinux.org-524b.rsa.pub
    -----BEGIN PUBLIC KEY-----
    (...)
    -----END PUBLIC KEY-----
init_config_files:
  - path: /etc/passwd
    content: |-
      root:x:0:0:root:/root:/bin/ash
      daemon:x:2:2:daemon:/sbin:/sbin/nologin
      (...)
  - path: /etc/shadow
    content: |-
      root:$6$UmJDHY...25/:18206:0:::::
      daemon:!::0:::::
      (...)
  - path: /etc/group
    content: |-
      root:x:0:root
      daemon:x:2:root,bin,daemon
      (...)
```

6.5.5 Solution to Problem 4: Quorum

An adversary might leverage administrative privileges to drop network traffic to certain hosts. In particular, she might prevent ROD from accessing the original repository, forcing ROD to rely on a mirror serving outdated software packages.

As specified in §6.3, we assume that the majority of repository mirrors are available and provide the latest snapshot of the original repository. ROD does not trust any individual mirror. Instead, it reads *2f+1* mirrors and only relies on the information that matches responses of at least *f+1* mirrors. Importantly, ROD requires a quorum only when reading the meta-

data index. The packages can be downloaded from a single mirror because their integrity is verifiable using the metadata index.

To allow different organizations to specify individual security requirements (*i.e.*, which mirrors to use, which package creators to trust) and to provide custom initial operating system configuration (*i.e.*, initial users, groups, and passwords), ROD accepts security policies. Listing 6.1 shows an example of such a security policy. The format permits defining a list of mirrors (Listing 6.1 lines 1-16) and a list of trusted package signers (Listing 6.1 lines 17-25). The package signer is a developer or a build system (*e.g.*, continuous integration and continuous deployment) that builds, signs, and deploys packages to the original repository.

ROD enforces the security policy by publishing only software packages in versions offered by the majority of available mirrors and only created by trusted entities. The policy could be extended to support a private/closed variant in which an operating system owner can specify a subset of supported software packages by specifying whitelist/blacklist of packages.

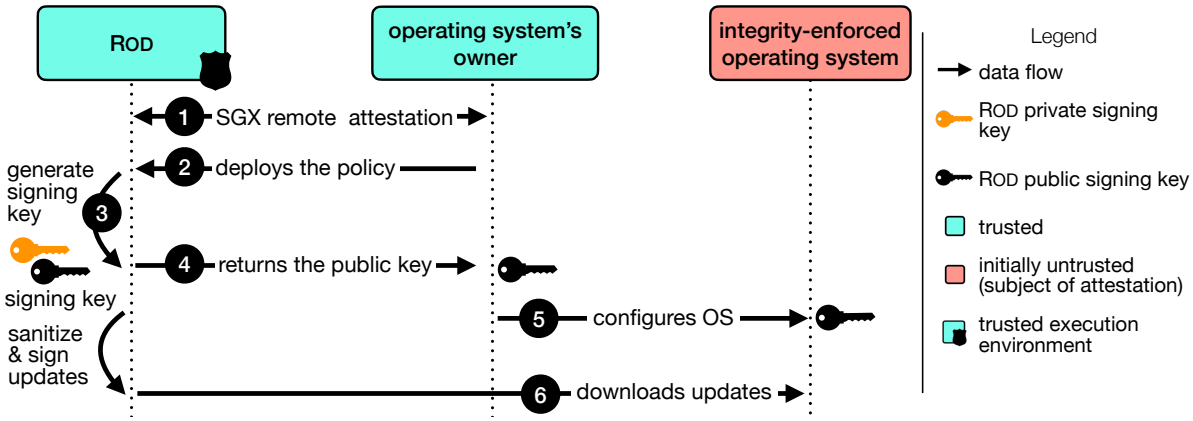


Figure 6.7. The protocol of distributing the public portion of the signing key, which can be used to verify the authenticity of the software packages.

Figure 6.7 shows how an organization can deploy a security policy to ROD. First, it establishes trust with ROD (❶) using SGX remote attestation protocol [133], which permits ensuring that ROD executes inside an enclave on the genuine Intel CPU. Then, it uploads the security policy (❷), causing ROD to generate a new signing key (❸), to store the security policy, and to return the public portion of the newly generated signing key (❹). Finally, the public key is distributed to all integrity-enforced operating systems and integrity monitoring systems (❺). At this point, the operating system accepts sanitized software packages (❻), and the integrity monitoring system accepts integrity measurements of files digitally signed by ROD. In more detail, the integration between integrity monitoring systems and ROD consists of adjusting integrity monitoring systems configuration to trust ROD signing key. Hence, integrity monitoring systems accept integrity measurements signed by ROD. ROD returns the signing key during the repository initialization (§6.6.2) triggered by the operating system owner (Figure 6.7).

6.6 Implementation

We developed ROD in Rust, a programming language that ensures memory safety [180]. We rely on the external Rust libraries, *i.e.*, Hyper [108], Rustls [134], to build the represen-

tational state transfer (REST) application programming interface (API) [66]. We use a Rust-based crypto library ring [27] to issue digital signatures. We use SCONE Rust cross-compilers [259] to execute ROD inside an SGX enclave. ROD is about 3.3k source lines of code, excluding external libraries.

We rely on SGX because it provides the following properties: *confidentiality* to protect the signing keys, *integrity* to protect the sanitization process, and *attestation protocol* to remotely ensure ROD integrity during the policy deployment. Alternative TEEs [169, 116, 87, 184] providing similar functionality might be considered but the threat model should be carefully adjusted according to TEE-specific implementation. For example, TEEs relying on late-launch technologies [116, 87, 184] must assume trusted link between CPU and TPM [272, 271], while others, like Keystone [169], must assume trusted boot process.

6.6.1 Supported Package Formats

Our prototype implementation of ROD supports *apk* packages used by Alpine Linux. We selected Alpine Linux because it is a popular security-oriented Linux distribution that minimizes the amount of software required to run the operating system. It is an important property for systems relying on trusted computing. In the future, we plan to add support for other formats (*i.e.*, deb, rpm) used by other Linux distributions.

6.6.2 Repository Initialization

ROD can be executed in the cloud and is operated by a cloud provider, who is responsible for correct hardware initialization, installation of the operating system, and ROD execution. The cloud provider exposes the hostname on which ROD API is accessible by his clients.

Multiple clients share a single ROD instance. Each client deploys a policy to create his individual, logically separated, software repository within the ROD instance. For each new repository, ROD, which runs inside an SGX enclave, generates a unique repository identifier and a unique signing key. The identifier and the public portion of the signing key are returned to the client as a response to the policy deployment request issued via https. Each client accesses his repository via the REST API after providing the identifier. By verifying the digital signature of the package, the client ensures that the package conforms to his requirements defined inside the policy.

6.6.3 Package Sanitization

We define package sanitization as an operation consisting of the following steps: verifying package integrity and authenticity, extracting files from the package archive, modifying the installation scripts (see §6.5.2), issuing digital signatures to all files inside the package, updating the metafile, and recreating the package. ROD issues digital signatures using the signing key generated during the policy deployment.

The digital signatures are stored inside portable archive exchange (PAX) headers [115] of the tar archive [118], which is logically equivalent to the package. The modern versions of tar extractors (*e.g.*, GNU tar [71]) transparently copy the specific PAX headers' value into the extended attributes in the filesystem. Before opening a file, Linux IMA scans extended attributes and includes the digital signature inside a dedicated file (IMA log). Consequently, the

monitoring systems read the measurement report and the IMA log. They check the integrity of every file measured by the IMA by verifying its digital signature included inside the IMA log.

6.6.4 Operating System Configuration

Software repositories include information about software packages sizes and hashes inside the repository metadata index to mitigate the endless data attack and the extraneous dependencies attack [32]. Operating systems read the package size and its hash from the metadata index to ensure they download the file of the expected size and contents. Because of that, when an operating system requests ROD to return the metadata index for the first time, ROD downloads and sanitizes all packages listed in the upstream metadata index. Then, ROD generates a new metadata index that matches the sanitized packages and returns it. Although the first metadata index generation is time-consuming, subsequent requests require ROD to sanitize only packages that have changed on the upstream mirrors, since the previous read.

Each integrity-enforced operating system must be reconfigured to use the ROD repository instead of mirrors. Moreover, the operating system must trust the packages signed by ROD; thus, the public portion of the signing key must be added to the list of trusted signers. This reconfiguration can be done automatically using configuration management systems such as Puppet [119] or Chef [117].

6.6.5 Package Caching

A slow read of software updates increases the vulnerability window for the time of check to time of use (TOCTOU) attack, where an adversary exploits the existing vulnerabilities until the security patches become available in the repository. In the case of ROD, this time is increased by the sanitization process (see §6.5.2) and the time required to read the majority of available mirrors (see §6.5.5).

To minimize the vulnerability window for the TOCTOU attack, ROD uses a local file system to cache the already sanitized packages, including the metadata index. ROD detects the outdated software packages each time ROD reads the new metadata index from the upstream mirrors. Consequently, ROD invalidates the metadata index, downloads the new version of the package, sanitizes it, and stores the new version inside the cache.

An adversary might tamper with the cache by reverting software packages and the metadata index to outdated versions. To mitigate the attack, ROD stores metadata indexes (the latest one read from upstream mirrors and the one reflecting the already sanitized packages) inside its memory, which integrity and freshness are guaranteed by SGX. ROD uses the first metadata index to check which software packages changed in the upstream mirrors. It uses the second metadata index to verify that the package read from the cache (untrusted disk) has not been rollbacked, before returning it to the operating system.

However, the data stored inside ROD memory is lost as soon as ROD is shutdown, for example, due to the operating system restart. To preserve the metadata indexes across ROD restarts, we extended ROD implementation with support for TPM monotonic counter (MC) [90]. After generating the metafile, ROD increases the MC value and uses SGX sealing [9] to store the metadata indexes together with the MC value on the disk. The SGX sealing, and its revert operation unsealing, uses a CPU- and enclave-specific key. Hence, only the same enclave running on the same CPU can unseal the previously sealed file. After the restart, ROD

unseals the metadata indexes from the disk together with the MC value and verifies that the unsealed MC value matches the current MC value.

6.7 Evaluation

In this section, we evaluate ROD to answer the following questions:

- What is the overhead related to the package sanitization?
- What are the performance limitations incurred by running ROD inside an SGX enclave?
- What is the cost of tolerating compromised mirrors?

Testbed. Experiments executed on a rack-based cluster of Dell PowerEdge R330 servers equipped with an Intel Xeon E3-1280 v6 CPU, 64 GiB of RAM, Samsung SSD 850 EVO 1 TB. All machines have a 10 Gb Ethernet network interface card (NIC) connected to a 20 Gb/s switched network. The support for SGX is turned on; the hyper-threading is switched off. We statically configured SGX to reserve 128 MB of RAM for the enclave page cache (EPC) [45]. The central processing units (CPUs) are on the microcode patch level 0x5e. We ran Alpine Linux 3.10 with enabled Linux IMA.

6.7.1 Package Sanitization Overhead

The sanitization process directly influences the software update process, *i.e.*, time after which software updates are visible by the operating system and the latency taken by the operating system to download the update. For that reason, we ran experiments in which we instrumented the sanitization process to measure its impact on packages from the main and community repositories of Alpine Linux. The results are based on a 20% trimmed mean from six independent experiment executions.

How much time does it take to sanitize all packages?

From the operating system perspective, low repository initialization time results in faster delivery of software updates. Therefore, we calculated the time requires to create a new repository, *i.e.*, to download and to sanitize all packages. In the case of packages updates, this time is expected to be significantly lower because ROD would have to download and to sanitize just a small amount of packages.

Table 6.3. Time required to initialize a repository. We assume two scenarios. In the optimistic one, ROD has access to a copy of packages stored in a cache. In the pessimistic one, during the policy deployment, ROD must download all packages from the original repository.

Time		
Pessimistic	Optimistic	Operation
17 min	0 min	Download packages
< 1 min	< 1 min	Policy deployment
13 min	13 min	Sanitize packages
30 min	13 min	Total

Table 6.3 shows the time taken to establish a new repository, assuming two scenarios. In the optimistic scenario, which takes about 13 min, ROD has access to pre-fetched packages,

which are available, for example, pre-fetched by a service provider. In the pessimistic one, which takes about 30 min, ROD additionally downloads original packages (about 3 GB of data) from upstream repositories. We argue that the download time can be greatly reduced by enabling parallel downloading. This performance improvement is left as part of future work.

What are the main factors driving the sanitization time?

ROD sanitizes all packages provided with a software update, thus introducing a delay in how fast the operating system receives the update. Therefore, it is important to understand the main drivers controlling the sanitization time.

Table 6.4. Spearman rank correlation coefficients (ρ) relating the package-specific properties and sanitization-specific operations. The corresponding p values are indicated by regular font in grey fields ($p < 0.05$), bold font in grey fields ($p < 0.001$); fields with regular font indicate $p > 0.05$.

	Number of files	Package size
Archive, compress	.46	**.61**
Check integrity	**- .62**	**- .93**
Generate signatures	**.69**	**.03**
Modify scripts	-.27	**- .33**

Table 6.4 shows the correlations between package-specific properties (*i.e.*, number of files inside a package, package size) and the proportional time contribution of certain components of the sanitization time. We observe a strong positive correlation ($\rho = 0.61$) between the archive processing time and package size, which indicates that the archive, compression and decompression algorithms take more time to process bigger archives. Also, we observe a strong correlation ($\rho = 0.69$) between signatures generation and the number of files inside a package. It confirms the intuitive expectation that in packages containing many files, the signature generation becomes a dominant factor of the sanitization time. Furthermore, we explain that a strong negative correlation ($\rho = -0.93$) between checking the package integrity and package size shows that the time required to check the package integrity becomes negligible for bigger packages because other operations (*i.e.*, signature generation, archive, compression and decompression) become the dominant factors. All in all, we anticipate that the sanitization time is mainly driven by (i) extracting files from a package and compressing them again into a package, (ii) issuing digital signatures.

How much time does it take to sanitize a package?

To better estimate time which ROD requires to expose an update, we examine the time it takes to sanitize individual packages. Figure 6.8 shows the relationship between sanitization time and package-specific properties, such as the package size and the number of files inside the package. The sanitization time is not evenly distributed; it changes from 11 ms (50th percentile), 36 ms (75th percentile), 422 ms (95th percentile), to 30 sec (100th percentile).

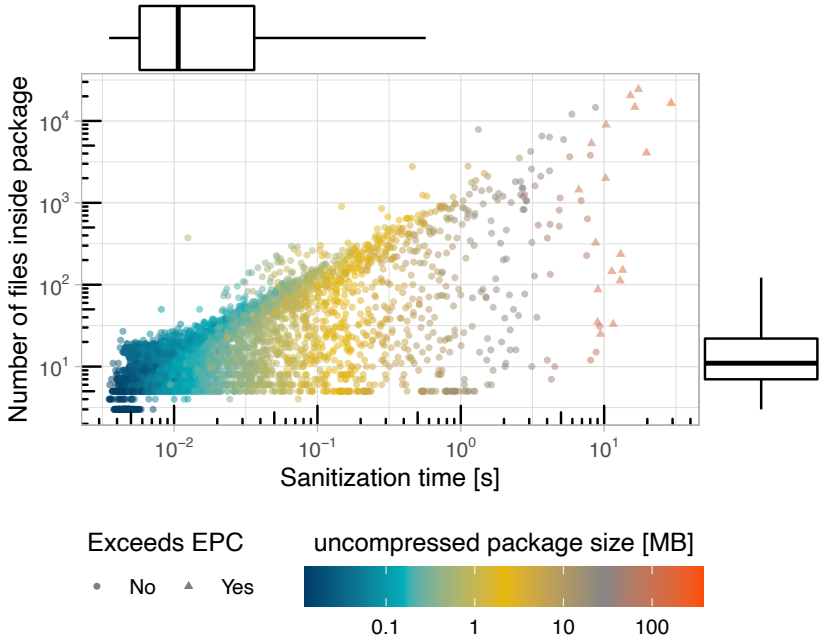


Figure 6.8. Time required to sanitize a package, depending on the number of files and size. Color represents package size after decompression. Packages which size exceeds the EPC are marked as ▲. Boxplots indicate 5th, 25th, 50th, 75th, and 95th percentile.

What is the impact of sanitization on the repository size?

Repository size is the sum of all packages served by the repository. The higher the size, the more resources (*i.e.*, disk space, bandwidth) are utilized. It not only increases the maintenance costs but also increases the latency because the operating system requires more time to download packages.

Figure 6.9 shows that the package sizes increase when compared to the original package size and the number of files located inside the package. In particular, the sanitization process increases package size by 12%, 27%, and 76% in 50th, 75th, and 95th percentile, respectively. Packages with many small files suffer most from sanitization because the sizes of file signatures (each signature is 256 bytes) constitute a dominant part of the total package size. However, the total repository size increases only by 3.6%, from 3000 MB to 3110 MB.

Does the caching decrease the latency of package download?

ROD implements caching to decrease the latency of accessing sanitized packages; it stores on the disk the original version of the package (the one fetched from upstream and not yet sanitized) and the sanitized one. We ran an experiment in which we measured how much time does ROD require to respond to a download request, assuming three scenarios: *(i)* only the original packages are cached, (*Original*), *(ii)* both original and sanitized packages are cached (*Sanitized*), and *(iii)* packages are not available in the cache (*None*).

In the first scenario, ROD downloads packages from an official Alpine mirror located on the same continent (an average network latency 26.4 ms). In the last two scenarios, ROD reads packages from the local disk. In each scenario, we requested ROD to return every

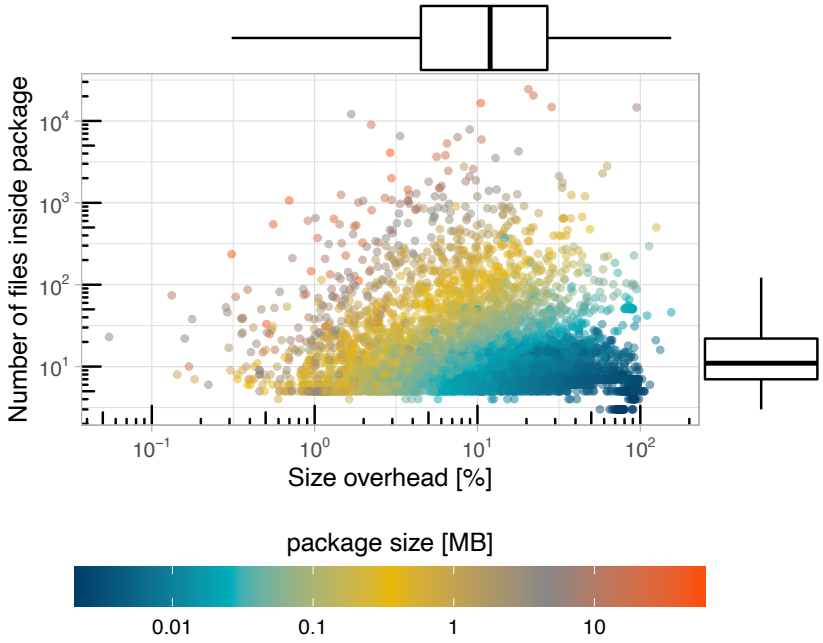


Figure 6.9. Increase of package size caused by sanitization, depending on the number of files inside the package. Color represents size of a package (files are compressed into a single archive). Boxplots indicate 5th, 25th, 50th, 75th, and 95th percentile.

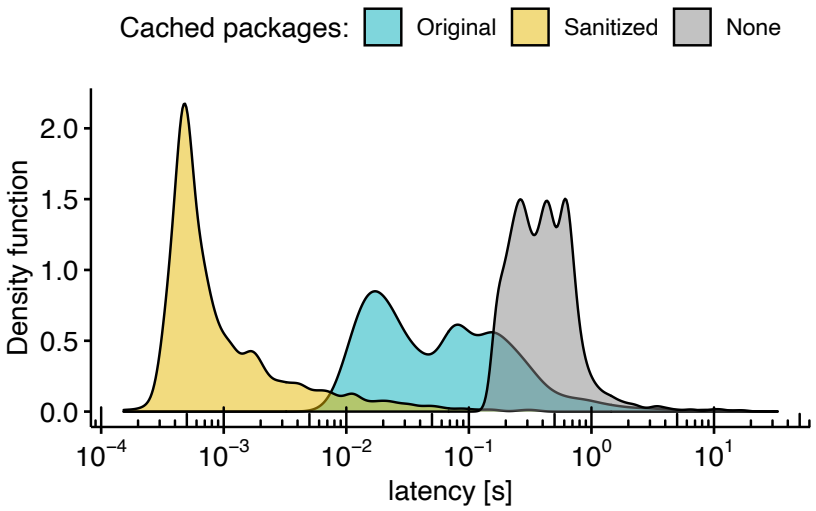


Figure 6.10. Comparison of package download latencies for scenarios in which ROD has access to original packages in the cache (*Original*), has access to already sanitized packages (*Sanitized*), and does not have access to any cached packages (*None*).

package available in the upstream Alpine repository sequentially. We calculated the latency of downloading each package as a 20% trimmed average from five repeated downloads.

Figure 6.10 shows distributions of package download latencies for the scenarios mentioned above. Caching the sanitization results decreases the average download latency 129× when compared to the scenario where ROD runs without cache. We anticipate that the la-

tency variation (0.37 ms) is mainly caused by accessing the cache (*i.e.*, reading packages of different sizes) and verifying packages integrity after reading them from untrusted storage.

Similarly, caching the original packages decreases the average download latency 2.7× when compared to the scenario where ROD runs without cache. This is mostly the result of a faster read of a package from the local disk than from a remote mirror accessed by the network.

What is the end-to-end latency of installing an update sanitized by ROD?

Installation of a software update takes a considerable amount of time because a package manager must download and verify the update, prepare the system for the new package version (check dependencies, lock installed packages database), unpack the new software package, launch installation scripts, copy files, set permissions, and finally clean the filesystem from no longer necessary files. In this experiment, we check the end-to-end latency of installing an update, which consists of sanitized packages or native Alpine packages. We measure the update installation latency for more than 5000 packages cached in a repository, *i.e.*, ROD serves sanitized packages from the cache. Before launching the experiment for each single package, we install the package, and then we tamper with the operating system configuration to pretend the installed package is outdated. We do it by modifying the package version number and its integrity hash stored in the file-based database used by Alpine Linux to store information about installed packages. Before measuring the next package, we uninstall the previously measured package from the operating system.

Figure 6.11 shows the experiment results in which we use two repositories, ROD and Alpine mirror, located in the same data center. We assume differences between network latency in both setups to be negligible. An average update installation latency is 141 ms and 110 ms for ROD and Alpine mirror, respectively. The higher latency observed when installing sanitized

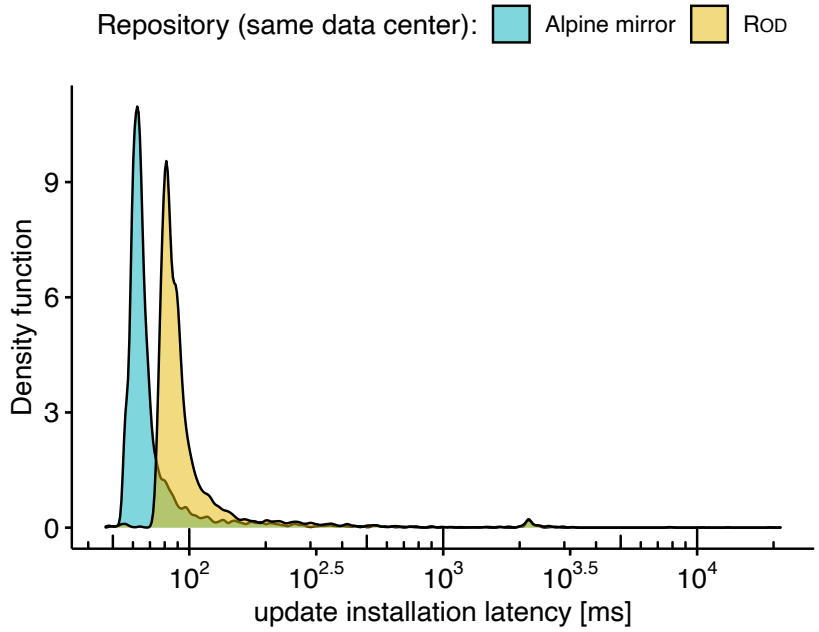


Figure 6.11. End-to-end latency of installing software updates.

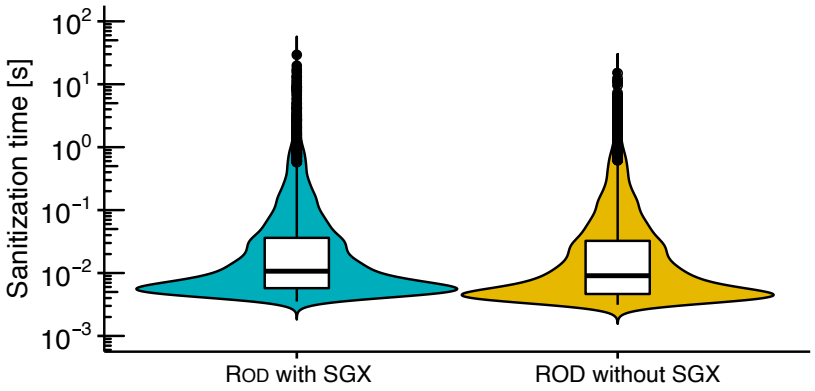


Figure 6.12. Violin plot showing comparison of sanitization times executed inside and outside of an SGX enclave. Boxplots indicate 5th, 25th, 50th, 75th, and 95th percentile.

packages is caused by installing digital signatures in the filesystem.

6.7.2 SGX Limitations

The current version of the SGX has limited memory, up to 128 MB for SGXv1. Applications that exceed this amount cause SGX to swap the memory leading to performance degradation. Hence, we address the question of:

What is the performance overhead of running ROD inside an SGX enclave?

To answer this question, we observe that the package sanitization is the most memory-consuming operation because ROD extracts and manipulates the package completely in the memory. For that reason, we executed ROD without SGX to measure the processing time of all available packages.

Figure 6.12 shows the comparison of packages sanitization times executed inside and outside an SGX enclave. We observe a minor overhead of executing inside SGX; 1.18× at 50th percentile, 1.12× at 75th percentile, and 1.16× at 95th percentile. However, at the top 5 percentiles that represent packages with sizes exceeding EPC, the SGX overhead increases to 1.96× because of EPC paging. The total sanitization time required to process all packages in the repository increases from 9.5 min to 13.6 min (1.43×) when running ROD inside an SGX enclave. In the future, SGXv2 might be used to overcome the limitation of the EPC memory, causing the largerst packages to be sanitized faster.

6.7.3 Tolerating Compromised Mirrors

What is the overhead of mitigating compromised mirrors?

In this experiment, we measured the latency in which ROD (running in Europe) returns the metadata index depending on the number of mirrors defined in the policy and their geo-

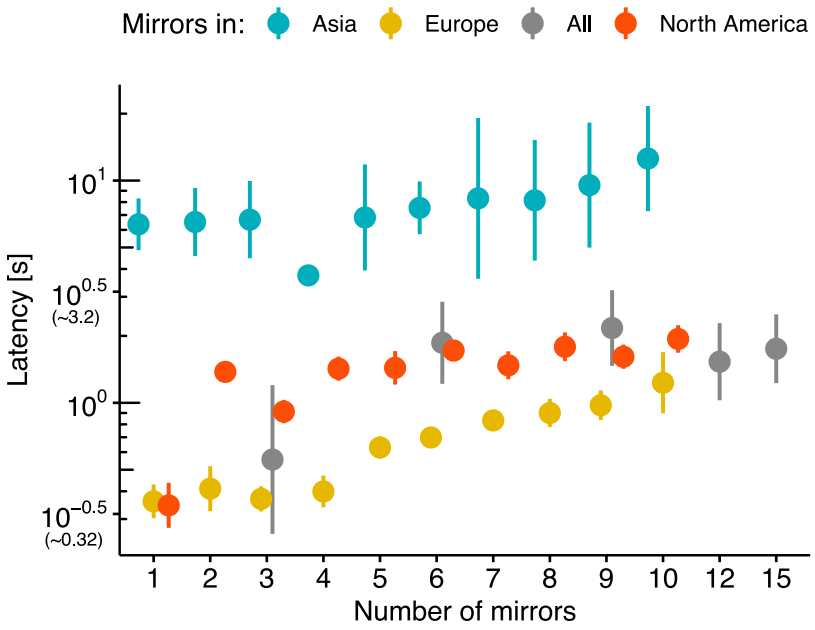


Figure 6.13. Latency of downloading the repository index from ROD, depending on the number and location of mirrors. ROD instance is deployed in Europe.

graphical locations. We were increasing the number of mirrors from one (default setting currently used by operating systems) to ten instances. To detect possible malicious responses, ROD used only the repository index which equal copies were returned by the majority of mirrors. For that, ROD calculated digests of indices to compare them among each other. We divided the experiment into four scenarios. In each scenario, ROD uses official Alpine mirrors located on different continents, *i.e.*, Asia, Europe, North America, and their combination (*All*). In each scenario, we calculated a 10% trimmed latency average from 20 consecutive requests.

Figure 6.13 shows that the latency of downloading the metadata index depends on the number and location of mirrors. ROD returns the metadata index in less than 400 ms for up to five mirrors on the same continent. In the case of 10 mirrors, ROD returns the metadata index in less than 1.2 sec. We observed higher latency when using mirrors located on different continents, mainly due to higher network latency.

The last scenario (*All*) shows that the latencies measured when mirrors are evenly distributed across three continents are similar to the latencies measured when using mirrors located only in North America. It is a result of ROD implementation; ROD contacts the fastest $f + 1$ mirrors, and, in case they present different metadata index, it contacts additional mirrors until reaching the quorum ($f + 1$ responses are the same). Therefore, mirrors in Europe and North America were preferred, and ROD latency depends on the slowest selected mirror.

It is the responsibility of the ROD clients to decide on the tradeoff between security and performance. The experiment shows that even when specifying nine mirrors distributed across different continents, ROD returns the metadata index in about 2.2 sec.

6.8 Related Work

Given the importance of software updates, a plethora of works have been proposed to ensure the security of software update systems [170, 69, 193, 286]. Typically, they aim to protect the updates using cryptographic signatures and transfer them to targets via secure connections. The critical aspect of these approaches is how to protect the signing keys because their leakage compromises the update process.

The Update Framework (TUF) [69] addresses the problem by assigning different roles for accessing specific signing keys, raising the bar for an adversary to get in possession of all keys. Unfortunately, TUF requires an online project registration; thus it cannot protect a community repository against several attacks, such as delivering arbitrarily modified packages. Diplomat [162] overcomes the shortcoming of TUF by dividing signing keys into offline and online keys. The online keys are used to provide fast package signing, a feature required in community repositories. Only online keys are leaked in the case of a repository compromise, which is a manageable problem since they can be easily revoked and the repository with new online keys can be regenerated using well-protected offline keys. CHAINIAC [193] provides mechanisms to secure the entire software supply chain. Developers create Merkel trees defining software packages with their corresponding binaries. To approve the package release, they sign and submit the trees to co-signing witness servers, which verify the signatures from developers as well as the mapping between the sources and the binaries. This mechanism relies on the blockchain technology, which permits the maintenance of the history of the releases but it increases the system's complexity. With a similar goal but reduced complexity, *in-toto* [252] offers a mechanism to ensure the integrity of the software supply chain cryptographically. It enables users with the integrity verification of the whole software supply chain. However, CHAINIAC, *in-toto*, and TUF do not consider the case that the target systems are under the protection of trusted computing mechanisms. Thus, they do not protect against integrity violations caused by software updates. Recently, KShot [286] introduced a secure kernel live patching mechanism to fix security vulnerabilities. KShot makes use of system management mode and SGX to perform the patching process without trusting the underlying operating system securely. Similarly, ROD leverages SGX to protect the software update patching mechanism (sanitization), but ROD also ensures that software updates do not break the operating system integrity. We selected Intel SGX to implement ROD since it has become available in clouds [141, 85], ported many of confidential cloud native applications including analytics systems [168, 167], key management system [88], and performance monitoring [156].

ROD follows the idea introduced by Berger et al. [23] to maintain a custom mirror with modified packages containing digital signatures. Unlike the previous work, ROD removes the mirror owner from the trusted computing base by protecting the signing keys using TEE. Also, ROD introduces the sanitization mechanism to enable the installation of packages containing installation scripts.

Several previous studies also considered various security aspects of the mirrors in software update systems [32, 150, 31]. Knockel et al. [150] indicated that man-in-the-middle attacks on third-party software are possible for open infrastructures. Fortunately, this can be handled by securing connections using modern TLS instead of outdated SSL technology. The Stork package manager [31] provided mechanisms to handle various attacks from malicious mirrors by dedicating the selective trust to users, *i.e.*, users specify which packages they trust to install. Mercury [161] addresses the rollback attacks on software packages [32, 18]

by maintaining a separated signed metafile at the package manager. However, Mercury did not address the problem of the first update in which a package manager cannot ensure the metadata index freshness. ROD tackles this problem by relying on the repository metadata index obtained from the majority of mirrors under the assumption that most mirrors are trustworthy.

6.9 Summary

In this chapter, we presented ROD, a trusted software repository that supports secure software updates of integrity-enforced operating systems. ROD is transparent to the existing implementations of package managers and software repositories. Importantly, it does not require changes to well-established distribution-specific procedures of creating software packages.

Our implementation supports 99.76% of the packages available in Linux Alpine main and community repositories. It can be hosted on-premises, *e.g.*, in the cloud, while maintaining strong security properties by running inside a TEE, enabling clients to define custom security policies, and permitting a minority of software repository mirrors to exhibit Byzantine behavior.

7 Security Configuration Management and Monitoring

High-assurance security systems require high-availability and fault-tolerance properties, and thus are designed and deployed as dependable systems. They are distributed over multiple computers consisting of different hardware and software, and are located in geographically distributed data centers to prevent a single point of failure. This complicates security management and monitoring. Consider as an example a system implementing the three-tier architecture. Such a system is separated into a presentation, an application, and a data tier. Each tier runs different components, possibly split into microservices, that differ in terms of software and configuration. For example, the presentation tier can be accessed directly by the end-user while the application tier (business logic) and the data tier (database and persistent storage) are accessible only by internal system processes. Such a design requires the application of a custom system and security configuration. For example, tiers will have different network and firewall settings to prevent unauthorized access to the business logic and data storage. Depending on the load, computers might be dynamically added or removed from the pool of computing resources on which the system's components are deployed. With the increasing number of computers, it becomes more and more difficult to efficiently configure and monitor the computer's integrity. In this chapter, we address the problem of how an application owner, or a respective security officer, could monitor that all the components of his distributed system conform with the security requirements?

Following existing approaches, such as Intel open cloud integrity technology (Intel CIT) [128] or Keylime [229], we propose ZORZA, a system simplifying management of security configuration and monitoring of remote computers. ZORZA is designed to be a practical system allowing for the quick adoption of the concepts presented in chapter 3, chapter 4, chapter 5, and chapter 6.

7.1 Contribution

We present ZORZA [13], a system simplifying integrity monitoring configuration and integrity verification of computers running high-assurance security systems. ZORZA has noteworthy

[13] In the Slavic mythology, Zorza is the goddess of the dusk.

advantages. First, it supports auto-discovery of computers, permitting scaling up and down the number of monitored computers without requiring manual intervention. Second, it performs automatic, recurrent integrity checking on behalf of the security officer, who gets automatically notified on integrity violations. Third, it simplifies security policy management, allowing the security officer to easily define trusted components, such as trusted operating systems, trusted TPMs, list of trusted firmware, and then combining them together to match the configuration of particular computers.

7.2 Design

Our objective is to provide an architecture that: (i) automatically detects new computers in the cluster and deploys corresponding security policies, (ii) allows the security officer to define security policies in a single place, (iii) recurrently verifies that monitored computers conform with the security policies.

Figure 7.1 shows the high-level overview of the ZORZA design. ZORZA comprises of three components: *(A)* The security configuration management and monitoring service is a web application that verifies the integrity of monitored computers, stores security policies, and notifies about integrity violations. *(B)* The database is persistent storage that stores security policies, a list of monitored computers, and an audit log of the monitored computers' integrity states. *(C)* Remote computers whose integrity is monitored. A security officer (D) is a person that is responsible for controlling the security of the provisioned computers. He defines security policies and takes action when policy violation is reported.

7.2.1 Discovery of Provisioned Computers

The security configuration management and monitoring service (SCMMS) is a standalone application running on a well-known hostname on which it exposes a representational state transfer (REST) application programming interface (API). The agent running on a newly provisioned computer sends an *auto-discover message* to the SCMMS. Like this, an initial connection between the agent and the SCMMS is established and the SCMMS informs the security

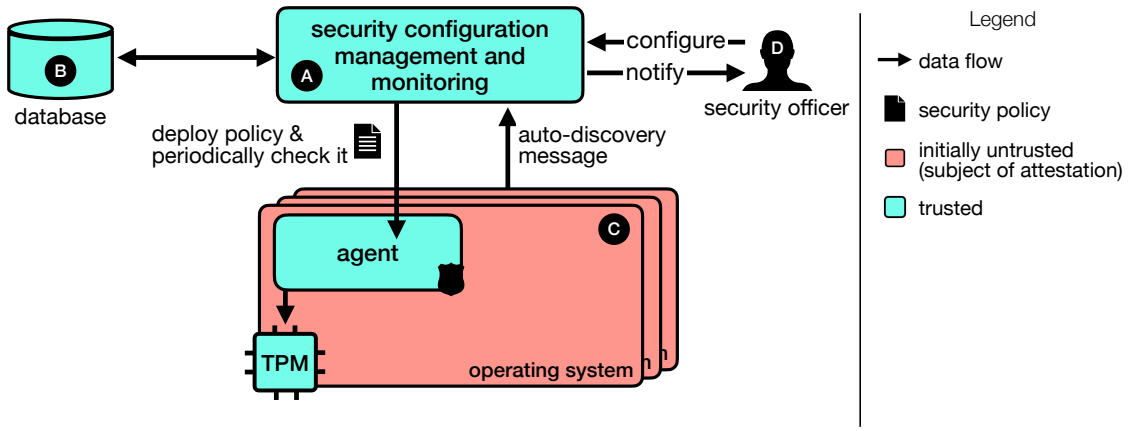


Figure 7.1. ZORZA design. The system owner interacts with the monitoring system via the security configuration management and monitoring service (SCMMS). The SCMMS stores security policies, the monitoring audit log, and performs integrity check of the monitored machines.

officer about the discovered computer. Typically, the security officer provisions computers and deploys the operating system with the agent using DevOps tools, such as configuration automation tools [119, 117], or it is automatically done by orchestration systems deploying and scaling applications [158].

7.2.2 Security Policy Configuration

The SCMMS reads the remote computer configuration and presents it to the security officer via a web user interface. The security officer inspects the initial computer's configuration and converts it into a new security policy, following the concept of trust on first use (TOFU), or assigns some of the existing security policies with the computer. Security policies, a monitoring audit log, and the list of monitored computers are stored in the database. We assume that SCMMS and the database are trusted, *i.e.*, run on trusted computers controlled by the security officer.

7.2.3 Policy Deployment and Monitoring

The security officer relies on the SCMMS to deploy security policies to agents running on monitored computers. The agent executes inside the SGX enclave that permits SCMMS to remotely attest to its integrity and delegate to it the remote computer's integrity checking. ZORZA offers good scalability because the computation-heavy tasks are done locally on each computer and not on a centralized server. Specifically, the agent verifies its proximity to the trusted beacon, reads the TPM quote and the IMA log comparing the read values to the deployed security policy. Consequently, the communication between the SCMMS and the agent is reduced to a single recurrent call (§3.6.4).

7.3 Implementation

We implemented ZORZA in the Python programming language. We used the Django [68] framework to build a web-based application utilizing the MariaDB database to store persistent data. We relied on Docker to containerize the ZORZA because it allows for quick configuration, testing, and deployment.

Figure 7.2 shows the implementation of ZORZA in the context of the work presented in this thesis. It leverages CHORS (§3) to provide implementation of the agent and trusted beacons. It further relies on TRIGLAV (§4) to provide the functionality of virtual computer runtime integrity monitoring and enforcement. PERUN (§5) provides a dedicated key management system to distribute keys to high-assurance security applications running on computers. Finally, ROD (§6) exposes sanitized software packages allowing for safe update of the operating system.

ZORZA extends the monitoring system presented in chapter 3 with

(i) a web interface simplifying policy management and deployment,
(ii) auto-discovery protocol allowing for simple addition of newly provisioned computers into the pool of monitoring resources,
(iii) notification tools informing the security officer about integrity violations.

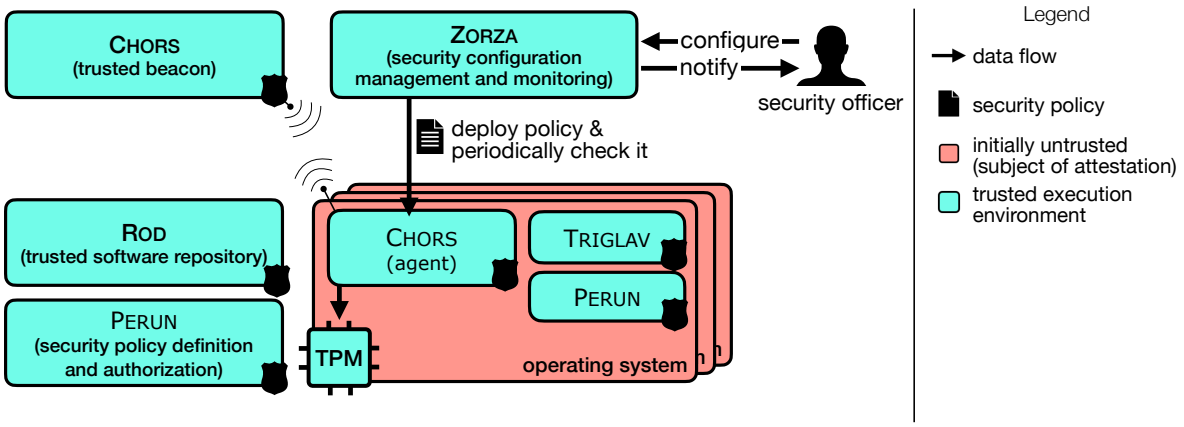


Figure 7.2. ZORZA implementation overview.

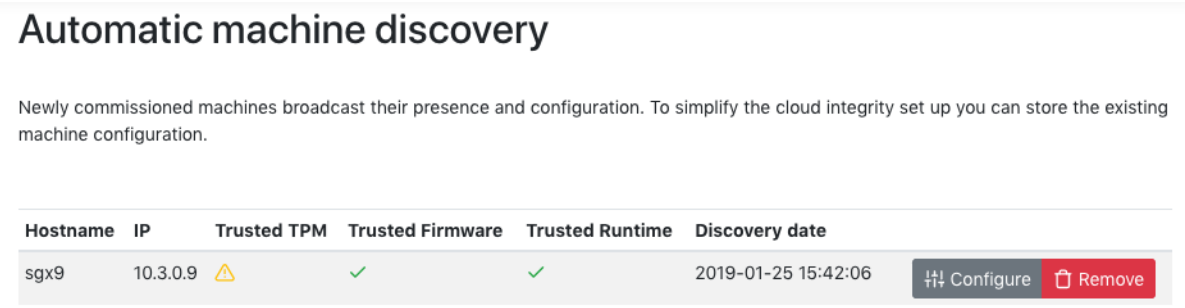


Figure 7.3. The CHORS's agent executes on newly provisioned machines. The agent contacts ZORZA, which runs on a well-known host. ZORZA collects infomation about the machine and allows the system owner to deploy specific policy to the machine and monitor it.

7.3.1 Auto-discovery

During the auto-discovery, the SCMMS fetches from the CHORS's agent the computer's configuration, *i.e.*, the TPM certificate issued by the TPM's manufacturer, the load-time integrity of the operating system, the operating system's runtime integrity measurements collected by IMA. Based on these information, the SCMMS matches the computer's configuration with the existing security policies stored in the database (Figure 7.3). For example, the TPM is verified based on the white list of TPM certificates or trusted manufacturers' certificate chains (Figure 7.4). Such functionality allows for the automatic addition of new computers to the monitored pool of computing resources without the burden of manual provisioning of every single computer.

7.3.2 Policy Creation

The security officer can predefine the security policies or can follow the concept of TOFU. To predefine security policies, he must define the list of trusted firmware in the form of static platform configuration registers (PCRs) white list, trusted operating system in the form of dynamic PCRs and a white list of application's measurements or a certificate allowing for verification of signed IMA measurements, and trusted trusted platform modules (TPMs) in the

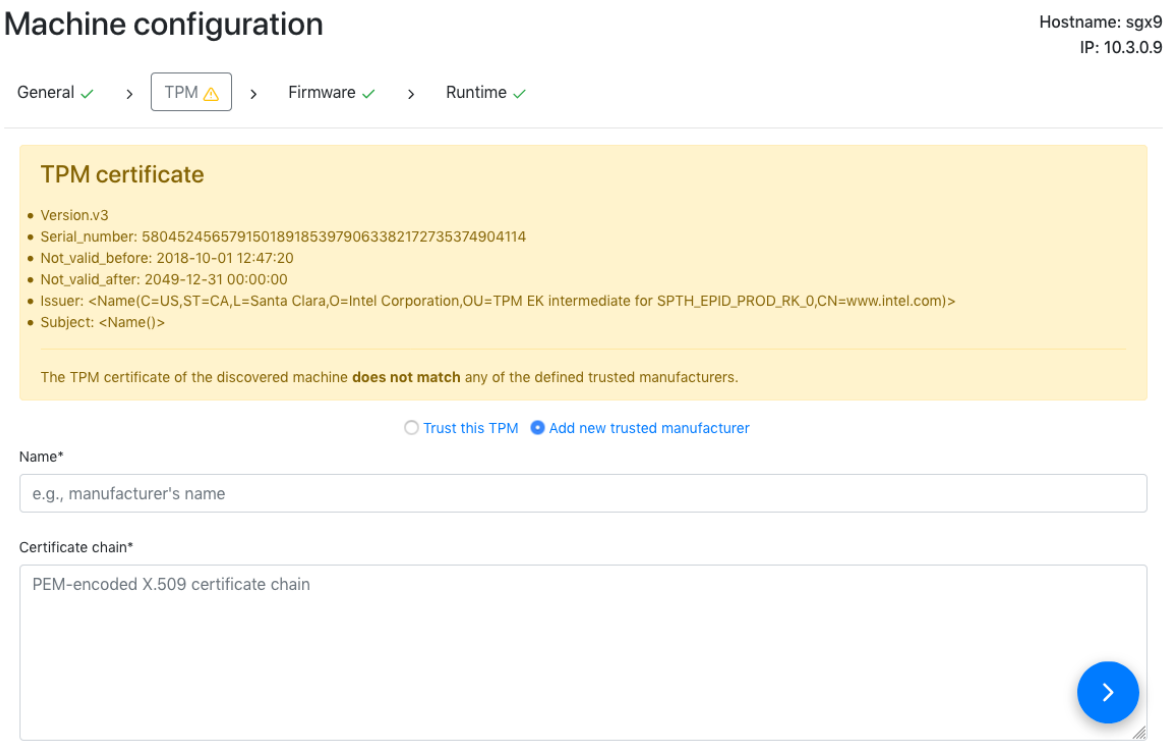
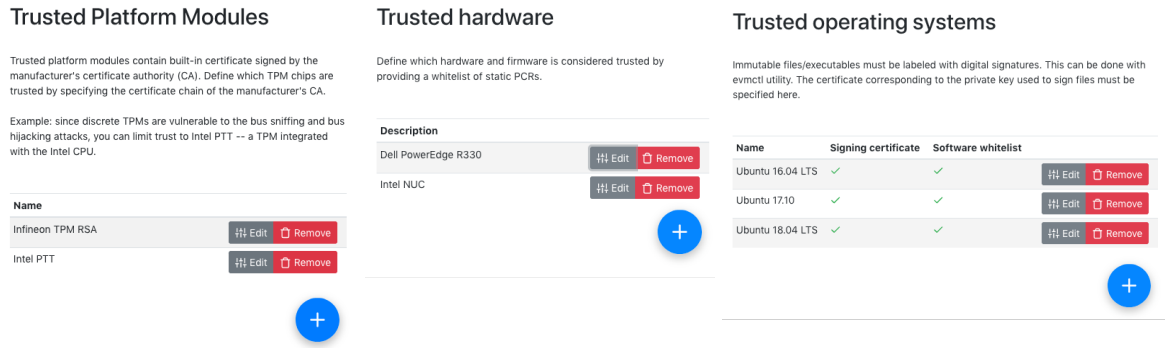


Figure 7.4. ZORZA simplifies defining machine-specific policy by providing a web-based configuration tool that automatically collects machine-specific information. The security officer can then use the collected configuration, define custom one, or use a configuration of other machines.



(a) Trusted platform module (b) Trusted hardware (c) Trusted operating system

Figure 7.5. A security officer defines a white list of trusted TPMs or TPMs' manufacturers, expected integrity measurements of firmware, and operating system load-time and runtime integrity measurements.

form of a whitelist of certificates or TPM's CA certificate chains (Figure 7.5). Trusted beacons are defined as a hostname on which they expose the distance bounding protocol (see §3.5.3) and a TLS certificate allowing to authenticate the trusted beacon (Figure 7.6).

The easiest way to create policies is to rely on the TOFU approach. The security officer can inspect the computer's configuration obtained during the auto-discovery and accept them as a trusted configuration, *i.e.*, he can request the SCMMS to create a new security policy based on the collected configuration. Other computers having the same hardware or the

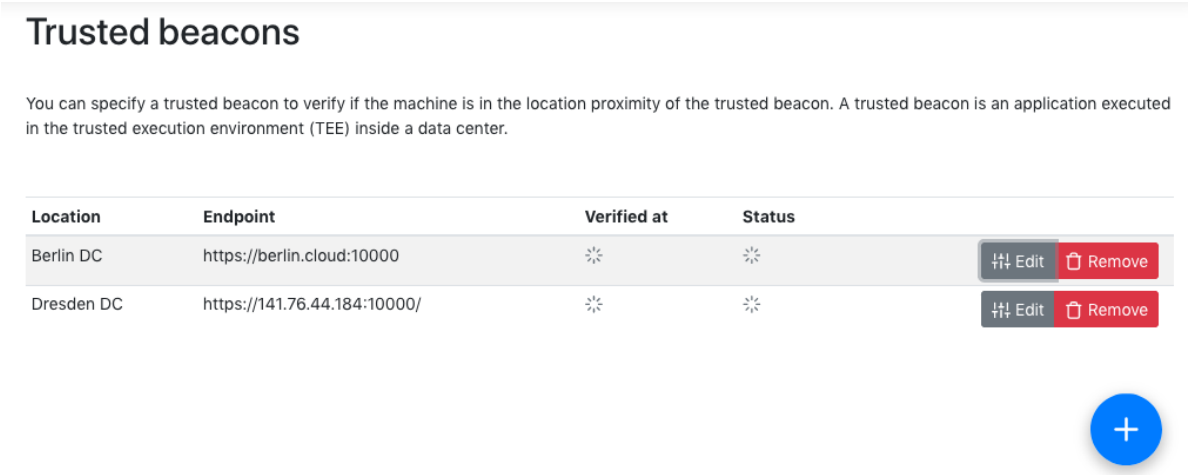


Figure 7.6. Trusted Beacon

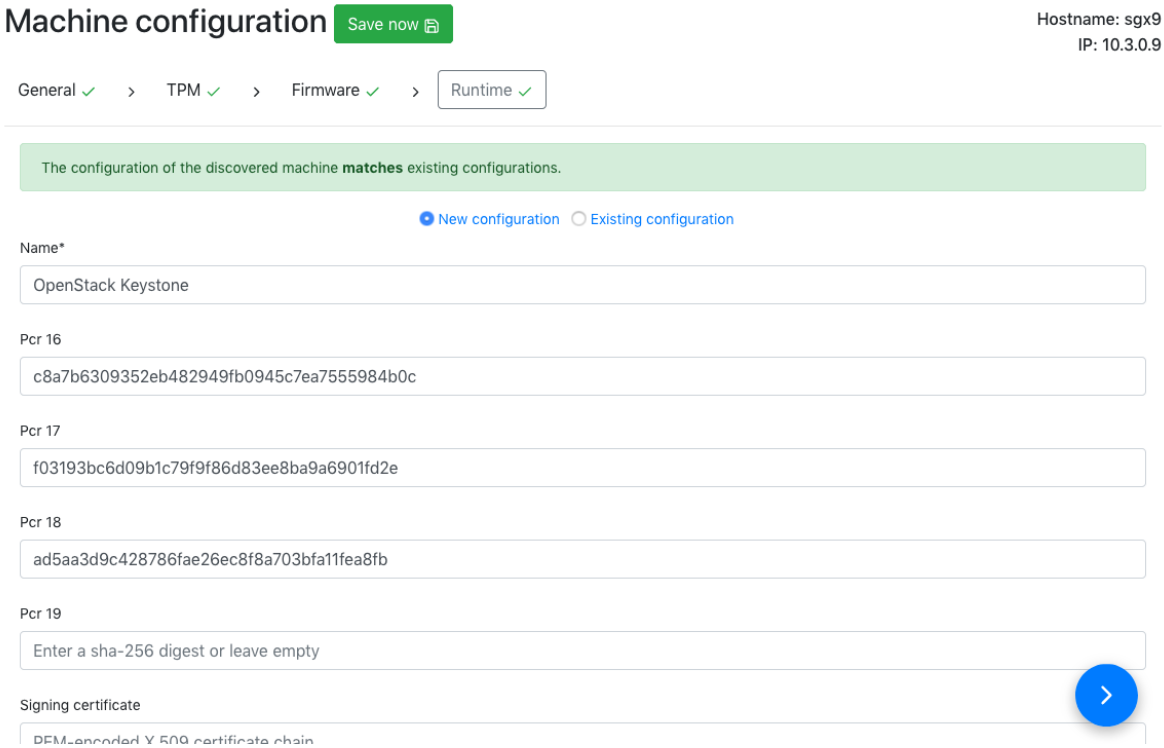


Figure 7.7. Machine runtime configuration.

same operating system will be automatically matched against the existing security policies (Figure 7.7).

7.4 Evaluation

To evaluate the ZORZA capability of detecting integrity violations, we established a setup where an adversary exploits a webserver misconfiguration to get remote access to a com-

puter. An adversary changes the root password in the remote operating system's configuration file by providing the webserver with malicious input.

7.4.1 Experiment Setup

The setup consists of a rack-based cluster of two Dell PowerEdge R330 servers connected via a 10 Gb Ethernet. The CHORS's agent and ZORZA run on different computers running Ubuntu Linux. The monitored computer runs an nginx [65] webserver exposing a PHP website. The website provides an HTML login form allowing its users to authenticate.

The operating system of the computer running webserver contains two vulnerabilities that are exploited by the adversary. Firstly, the nginx runs with root permissions, Listing 7.1. Secondly, the PHP script does not sanitize the input, which is eventually passed to the PHP's shell_exec command, Listing 7.2. The combination of these two vulnerabilities allows an adversary to mount a simple attack, tampering with the remote computer's integrity. Although the presented attack is a very simple attack implemented just for demonstration purposes, ZORZA detects attacks that modify configuration files, binaries, or execute untrusted software (see §3.3). Please note that in this setup, we do not rely on the integrity measurement architecture (IMA) enforcement mechanism, which would prevent these kinds of attacks by rejecting tampered files from being loaded to the memory.

The computer runs the CHORS's agent inside Intel software guard extensions (SGX) enclave. The agent performs runtime integrity verification by recurrently reading IMA events and comparing them to the policy deployed by the security officer via ZORZA. ZORZA runs on another computer. It recurrently queries the agent to check if the remote computer's integrity conforms to the policy.

Listing 7.1: Vulnerability 1: The webserver executed with too broad permissions

```
# ps aux
# ...
3507 root      0:00 nginx: master process /usr/sbin/nginx −c /etc/nginx/nginx.conf
3508 www       0:00 nginx: worker process
3540 root      0:00 {php−fpm7} php−fpm: master process (/etc/php7/php−fpm.conf)
3547 root      0:00 {php−fpm7} php−fpm: pool www
3548 root      0:00 {php−fpm7} php−fpm: pool www
# ...
```

Listing 7.2: Vulnerability 2: The script does not sanitize input

```
<?php
$login = $_GET['login'];
$password = sha1($_GET['password']);

if (strcmp($password, trim($correct_password_hash)) == 0) {
  echo
} else {
  echo 'Incorrect login or password';
}
?>
```

Listing 7.3: Malicious input: The malicious input allowing an adversary to tamper with the operating system configuration in order to gaining remote access

```
# source code:
shell_exec("cat /home/$login/password.txt");
# malicious input:
$login = "||pass='SOME_HASH'&&sed -I -e 's,root:[:]\+:,root:$pass:,' /etc/shadow ||
    echo ";
# executed command in runtime:
cat /home/||pass='SOME_HASH'&&sed -I -e 's,root:[:]\+:,root:$pass:,' /etc/shadow ||
    echo /password.txt
```

7.4.2 Experiment Scenario

The adversary requests via HTTP calls a webserver to render an HTML code of the login page. By checking different login and password configurations, he finds out that the script does not sanitize input because it outputs the received input to the output of the HTML code. Since such requests do not cause the webserver integrity violation, no integrity violations are reported by ZORZA.

The adversary prepares a malicious input that will cause the script to tamper with the operating system configuration, *i.e.*, the root password in the /etc/shadow file. Specifically, the adversary leverages the fact that the content of the login field input is directly passed to the shell_exec command and that the webserver runs with root permissions. This allows the adversary to execute arbitrary commands with root privileges. Listing 7.3 shows the malicious input crafted by the adversary and the resulting command executed by the script in a shell. By changing the SOME_HASH to a hash value corresponding to a password known to the attacker, an attacker can change the root password, gaining remote access to the computer via, for example, SSH. An adversary might directly leverage the possibility of running arbitrary commands to download and execute an exploit. Executing an exploit or any command-line tool that is not whitelisted in the security policy deployed by ZORZA, causes an integrity violation.

In our experiment, ZORZA returned the runtime integrity violation of the computer hosting the webserver as soon as the adversary modified the root password. Specifically, IMA measured the new content of the /etc/shadow file. The agent detected that the file content hash was not included in the policy whitelist and reported the policy violation to ZORZA, which notified the security officer about the incident. Depending on the use case, one might configure automatic incident response to respond to attack in real-time or configure IMA to prevent loading such a file, using IMA integrity enforcement mechanism (§2.2.5).

7.5 Related Work

The Intel CIT [128], the successor of OpenAttestation [124], is an open-source integrity monitoring system provided by Intel. It relies on tboot to establish a dynamic root of trust for measurements (DRTM) with Intel trusted execution technology (TXT) [87], and TPM to securely store platform measurements. Intel CIT integrates with OpenStack [202], to which it exposes the hosts' security properties allowing to group resources in trusted computing pools. Similarly to ZORZA, the Intel CIT's *trust agent* is deployed on each host to retrieve the

measurements from TPM, TXT logs as well as the operating system configuration. During the start up of the trust agent, a bash script *module_analysis.sh* is executed. It parses the output of tboot (using txt-stat command) to produce an XML formatted file stored in plaintext on the disk. The file is not protected against tampering and is regenerated only during the restart of the agent. TLS protected web services implemented in Java are executed on top of Jetty application server, exposing host measurements and configuration to the Open CIT central server. The host configuration is read from the operating system and pre-caches the txt-stat output. The measurements are obtained by executing the *tpm2_quote* command-line utility on each request to the POST */tpm/quote* web service. The authenticity, integrity, and freshness of the quote is verified by the central Open CIT server. Compared to ZORZA, Intel CIT performs integrity verification centrally and does not support the geolocation proximity verification and the TPM cuckoo attack detection. It does not allow tenants to verify that the acquired computing resources comply with the given security policy.

The IBM TPM attestation client-server (IBM ACS) [111] is an open source project of a sample Trusted Computing Group (TCG) attestation application written in C. It implements the TPM remote attestation in a centralized manner where the server gathers TPM measurements from hosts, compares them to the whitelist, and stores in a database. IBM ACS supports verification of the IMA measurements, TPM 1.2 and TPM 2.0. Unlike ZORZA, it peforms centralized integrity verification and does not protect against the cuckoo attack.

Keylime [229] also implements an integrity monitoring system that additionally integrates with a certificate authority (CA). The CA is used to revoke keys once the integrity violation is detected. Like this, other applications, such as IPsec, Pupper, or LUKS can rely on the CA without requiring to communicate with the TPM and other trusted computing technologies. However, unlike ZORZA, Keylime performs centralized integrity verification, does not verify the geolocation proximity of the monitored machines, and does not address the cuckoo attack.

7.6 Conclusion

In this chapter, we presented ZORZA, a security configuration management and monitoring system. It facilitates the management of security configurations, their distribution to remote computers, and automatic verification of compliance of these computers with defined configurations. ZORZA enables a simplified use of the concepts presented in the previous chapters of this thesis.

8 Conclusion and Future Work

Given that our societies depend more and more on digital services that execute security- and safety-critical operations and process privacy-sensitive data, I investigated how to establish technical assurance that these services execute securely. Specifically, I looked at how to attest that a high-assurance security system executes in a dedicated data center, on top of a trustworthy operating system, and isolated from other software. My approach combines two state-of-the-art solutions, *trusted computing techniques (TCTs)* and *trusted execution environment (TEE)*. I demonstrated that they are complementary, but their combination requires solving additional issues specific to each solution. I presented how to solve these issues and demonstrated the practicality of my approach by building prototypes, which I evaluated using real-world applications.

8.1 Summary of Results

My research contributed a framework dedicated to the secure execution of high-assurance security systems in remote execution environments. The framework implements novel techniques that provide technical assurance that high-assurance security systems execute in isolation from other software and run on trustworthy operating systems. More precisely, high-assurance security systems utilizing my framework maintain integrity and confidentiality guarantees of their code and data while ensuring the runtime integrity of the surrounding operating system. The solution follows assumptions of high-assurance security systems that require isolation of computing resources at the data center level.

I designed the framework focusing on the practicality, *i.e.*, the framework has been implemented with available security mechanisms and hardware. The framework supports state-of-the-art technologies (*i.e.*, Linux kernel with kernel-based virtual machine (KVM) virtualization [147]), uses existing off-the-shelf security mechanisms and hardware (*i.e.*, trusted platform module (TPM) [90], Intel software guard extensions (SGX) [45]), supports legacy applications without requiring source code changes, supports software updates, and induces low performance overhead (up to 6%). Here, I summarize each of the techniques introduced.

8.1.1 Cuckoo Attack Defense Mechanism

By relying on the properties of the TPM, dynamic root of trust for measurements (DRTM) [235], and SGX I designed a practical, deterministic cuckoo attack detection protocol. It enables trust to be established from the inside of the SGX enclave to the TPM chip, recording integrity measurements of the operating system. At the conceptual level, the protocol helps a process executed inside the TEE to learn about the trustworthiness of the surrounding operating system. This is particularly important in the case of high-assurance security systems because such systems depend on the operating system in terms of availability and confidentiality. Rasha Faqeh formally proved [203] the protocol to be immune to the cuckoo attack.

8.1.2 Integrity Monitoring and Enforcement Framework

I implemented an integrity monitoring and enforcement framework that controls the integrity of multiple services across multiple computers in the data center (chapter 3, chapter 7). In my design, a small piece of software executed inside the TEE acts as a trusted anchor (initially the only trusted piece of software) on a remote computer. It then extends trust to the secure element (a TPM chip), ensuring the lack of the cuckoo attack with the help of the defense mechanism mentioned above. The trust is eventually extended to the operating system level with the help of integrity monitoring and the enforcement mechanism, *i.e.*, integrity measurement architecture (IMA) [225, 89]. Crucially, the framework decentralizes the integrity attestation of machines by implementing a protocol that performs integrity checks on each monitored machine individually. This is only possible because of the TEE that guarantees the integrity of the attestation process.

8.1.3 Runtime Integrity-enforcement of Virtual Machines

I extended the protocol mentioned before to enforce the integrity of software executing inside virtual machines (chapter 4). The solution allows the virtual machine's owner to define software he trusts in the form of a security policy and delegate enforcement of this policy to a piece of trusted code protected by the TEE. My solution helps to better utilize computing resources by partitioning them into virtual machines while still enforcing the integrity of software used to establish the virtualized environment.

8.1.4 Multi-stakeholder Machine Learning Framework

I combined the previous work on secure key distribution [88] with integrity enforcement and attestation to allow the high-assurance security system's owner to trade-off between the security and performance in a multi-stakeholder environment (chapter 5). In my design, a trusted third party is responsible for enforcing access to the cryptographic keys. Depending on the use case, cryptographic keys are accessible only to a high-assurance security system executing inside TEE or inside a trustworthy operating system. In the latter case, the high-assurance security system has access to hardware accelerators at the cost of a larger trusted computing base. I argue that the trade-off is justifiable in the case of machine learning training computations that require access to large computing power.

8.1.5 Support for Software Updates of Integrity-enforced Operating Systems

I solved the problem that limited the practical application of TCTs for monitoring of operating system's integrity (chapter 6). I introduced the concept of *sanitization* in which a trusted third party (*i.e.*, an algorithm executing inside the TEE) transparently modifies software updates to certify the future system's state after predicting how it will look after the update is installed. From the security perspective, the design protects against freeze and replay attacks permitting a minority of software repository mirrors to exhibit Byzantine behavior. The evaluation of the prototype implementation showed that this approach supports 99.76% of the packages available in Linux Alpine main and community repositories.

8.2 Future Work

This thesis showed that despite the differences in designs and security guarantees of TEE and TCTs, both concepts could be combined together, solving issues specific to each technology and, eventually, increasing the security guarantees of high-assurance security systems. I identified potential directions that might expand my current work. I outline them in this section.

8.2.1 Policy-based Compliance Management

Depending on the country and the domain, high-assurance security systems are subject to different regulations. For example, German eHealth regulations [77] require the use of TEE on top of a trustworthy operating system, while other regulations might require just file integrity monitoring. Consequently, each such system requires the implementation of different security mechanisms and an expert-knowledge to ensure compliance with the regulations. The practical framework should then support a wide range of security mechanisms and the flexibility to select them accordingly to applicable regulations.

Existing approaches, such as OpenSCAP [215], provide policies and tools checking if the system configuration complies with specific regulations, like the payment card industry (PCI) data security standard (PCI-DSS) [206]. However, these approaches lack support for verifying if certain hardware-based technologies are used, what isolation levels for security-critical processes are provided, and if the attached hardware devices meet given standards. I anticipate that a production-ready solution should come with built-in policies, ideally semi-automatically extracted from regulations, that are then used by attestation engines to certify compliance of remote execution environments with given requirements.

8.2.2 Integrity Attestation of Mutable Files

Integrity measurement architecture (IMA) [225, 89] enables attestation of the runtime integrity of the operating system. The verifier, such as a remote entity checking the computer's integrity or a kernel's integrity-enforcement mechanism, can check if the file is legitimate by comparing the hash over its content to a hash representing an allowed content. However, this technique alone is not enough when a legitimate process creates or modifies a file because the new hash representing the file's content differs from the allowed value of the apriori known content.

I proposed a technique to deal with this problem in the context of software updates causing deterministic changes to the file system (subsubsection 6.5.2). However, this technique is not enough for a *general-purpose system* where temporal results are written and then read from a filesystem as part of the regular operating system's execution, because the integrity-enforcement mechanism would prevent such temporal files from reading causing undefined runtime behavior. A more relaxed approach, where system integrity is only recorded for attestation purposes but not enforced, would cause false positives in the integrity monitoring systems. I envision that future work solves this problem by focusing on certifying hashes of such temporal files from the inside of the TEE, leveraging causal order of integrity measurements, or providing a verifier with additional knowledge about the origin of such changes.

8.2.3 Availability Guarantees

Most of TEEs do not offer availability guarantees because, by their design, the enclave's lifecycle is under full control of an untrusted operating system and administrator. However, availability is a crucial security property of high-assurance security systems. The natural question that arises is how to provide the availability guarantees to the enclave without relying on the operating system? Could we bind the enclave's lifecycle with the policy presented to the CPU on the enclave initialization? Could CPU provide support for ensuring certain enclave's quality of service (QoS), like the number of resources or CPU time slots dedicated for the given enclave? Or, should we rather build TEE with the availability guarantee on the microkernel architecture?

8.2.4 Integration with SIEM

Security information and event management (SIEM), like QRadar [110] are industry-standard security solutions actively monitoring computer systems behavior that might indicate an attack. SIEM systems collect event data from various heterogeneous sources, such as file integrity monitoring and network monitoring, and correlate them with patterns indicating successful intrusion or attack attempts. The framework introduced in this thesis could complement SIEM because i) it mitigates certain attacks by enforcing the operating system's integrity, and ii) it collects file integrity measurements. I anticipate that future work might explore what exact security guarantees could be gained by combining SIEM, TEE, and TPM-based security mechanisms and if such combination would decrease the number of false positives?

8.2.5 Hardware-supported Virtual Machine Isolation

In chapter 4, I discussed TRIGLAV, a framework for virtual machine's runtime integrity monitoring and enforcement. However, to provide runtime integrity guarantees of a virtual machine, TRIGLAV must also establish trust in the hypervisor, enlarging the trusted computing base (TCB). In fact, VM-based trusted execution environment (TEE), such as Intel TDX [123] or AMD SEV [140] might reduce the TCB of the TRIGLAV design because these technologies provide integrity, freshness, and confidentiality of the virtual machine's memory in the face of an untrusted hypervisor. Future work might consider leveraging these technologies to improve the TRIGLAV design so that the tenant not only attests to the VM's load-time and runtime integrity but also gets the proof that the VM executes inside the TEE that isolates it from the hypervisor and operator.

Bibliography

[1] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, 2016.

[2] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep Learning with Differential Privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, 2016.

[3] Advanced Micro Devices. AMD64 virtualization: Secure Virtual Machine Architecture Reference Manual. *AMD Publication no. 33047, Rev. 3.01*, 2005.

[4] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. Obfuscuro: A commodity obfuscation engine on Intel SGX. In *Network and Distributed System Security Symposium*, 2019.

[5] Intel AI. Deep Learning Medical Decathlon Demos for Python. https://github.com/IntelAI/unet/, accessed on October 2021.

[6] Markuze Alex, Shay Vargaftik, Gil Kupfer, Boris Pismeny, Nadav Amit, Adam Morrison, and Dan Tsafrir. *Characterizing, Exploiting, and Detecting DMA Code Injection Vulnerabilities in the Presence of an IOMMU*. 2021.

[7] Alpine Linux Development Team. Alpine Linux - Small. Simple. Secure. https://alpinelinux.org/about/, accessed on October 2021.

[8] Amazon Web Services, Inc. Firecracker: secure and fast microVMs for serverless computing. http://firecracker-microvm.github.io, accessed on October 2021.

[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*, 2013.

[10] Arch Linux. Arch Linux: Arch build system. https://wiki.archlinux.org/index.php/Arch_Build_System, accessed on October 2021.

[11] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, David Goltzsche, Dave Eyers, Rueudiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[12] Will Arthur and David Challener. *A practical guide to TPM 2.0: Using the new trusted platform module in the new age of security*. Springer Nature, 2015.

[13] Aref Asvadishirehjini, Murat Kantarcioglu, and Bradley Malin. GOAT: GPU Outsourcing of Deep Learning Training With Asynchronous Probabilistic Integrity Verification Inside Trusted Execution Environment. *arXiv preprint arXiv:2010.08855*, 2020.

[14] BakerHostetler. International Compendium of Data Privacy Laws. https://towerwall.com/wp-content/uploads/2016/02/International-Compendium-of-Data-Privacy-Laws.pdf, accessed on October 2021.

[15] Guillaume Barbu, Hugues Thiebeauld, and Vincent Guerin. Attacks on java card 3.0 combining fault and logical attacks. In *International Conference on Smart Card Research and Advanced Applications*, 2010.

[16] BBC. Fukushima disaster: What happened at the nuclear plant? https://www.bbc.com/news/world-asia-56252695, accessed on October 2021.

[17] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATC' 05)*, 2005.

[18] Anthony Bellissimo, John Burgess, and Kevin Fu. Secure Software Updates: Disappointments and New Challenges. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Security (HOTSEC '06)*. USENIX Association, 2006.

[19] Stefan Berger. Libtpms: software emulation of a Trusted Platform Module. https://github.com/stefanberger/libtpms, accessed on October 2021.

[20] Stefan Berger. SWTPM - Software TPM Emulator. https://github.com/stefanberger/swtpm, accessed on October 2021.

[21] Stefan Berger, Ramon Caceres, Kenneth A. Goldman, Ronald Perez, Reiner Sailer, and Leendert van Doorn. vTPM: virtualizing the trusted platform module. In *Proceedings of the 15th Conference on USENIX Security Symposium (USENIX Security '06)*, 2006.

[22] Stefan Berger, Kenneth Goldman, Dimitrios Pendarakis, David Safford, Enriquillo Valdez, and Mimi Zohar. Scalable Attestation: A Step toward Secure and Trusted Clouds. In *IEEE International Conference on Cloud Engineering (IC2E 2015)*, 2015.

[23] Stefan Berger, Mehmet Kayaalp, Dimitrios Pendarakis, and Mimi Zohar. File Signatures Needed! *Linux Plumbers Conference*, 2016.

[24] Sandeep Bhatt, Pratyusa K. Manadhata, and Loai Zomlot. The operational role of security information and event management systems. *IEEE Security and Privacy (S&P)*, 2014.

[25] Jeremy Boone. Tpm genie: Interposer attacks against the trusted platform module serial bus. In *NCC Group Whitepaper*, 2018.

[26] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. DR. SGX: Automated and adjustable side-channel protection for SGX using data location randomization. In *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019.

[27] Brian Smith. Safe, fast, small crypto using Rust. https://github.com/briansmith/ring, accessed on October 2021.

[28] Milan Broz. LUKS2 On-Disk Format Specification, Version 1.0.0. In *LUKS documentation*, 2018.

[29] Kevin R.B. Butler, Stephen McLaughlin, and Patrick D. McDaniel. Rootkit-resistant disks. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS '08)*, 2008.

[30] Fabiana Cambricoli. Nova falha do Ministerio da Saude expoe dados pessoais de mais de 200 milhoes de brasileiros. https://saude.estadao.com.br/noticias/geral,nova-falha-do-ministerio-da-saude-expoe-dados-pessoais-de-mais-de-200-milhoes,70003536340, accessed on October 2021.

[31] Justin Cappos, Scott Baker, Jeremy Plichta, Duy Nyugen, Jason Hardies, Matt Borgard, Jeffry Johnston, and John H. Hartman. Stork: Package Management for Distributed VM Environments. In *Proceedings of the 21st Large Installation System Administration Conference (LISA '07)*, 2007.

[32] Justin Cappos, Justin Samuel, Scott Baker, and John H. Hartman. A look in the mirror: attacks on package managers. In *Proceedings of the 15th ACM conference on Computer and Communications Security (CCS '08)*, 2008.

[33] Justin Cappos, Justin Samuel, Scott Baker, and John H Hartman. Package management security. *University of Arizona Technical Report*, 2008.

[34] Chanandler Carruth. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html, accessed on October 2021.

[35] Marco Carvalho, Jared DeMott, Richard Ford, and David A Wheeler. Heartbleed 101. *IEEE security & privacy*, 2014.

[36] Cen, Schanwei and Zhang, Bo. Trusted Time and Monotonic Counters with Intel Software Guard Extensions Platform Services. Intel white paper, Intel, 2017.

[37] Somnath Chakrabarti, Brandon Baker, and Mona Vij. Intel SGX Enabled Key Manager Service with OpenStack Barbican. *arXiv preprint:1712.07694*, 2017.

[38] Dhiman Chakraborty, Lucjan Hanzlik, and Sven Bugiel. simTPM: User-centric TPM for Mobile Devices. In *28th USENIX Security Symposium (USENIX Security '19)*, 2019.

[39] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, 2013.

[40] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2019.

[41] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[42] William R. Claycomb. Detecting Insider Threats: Who Is Winning the Game? (MIST '15). In *Proceedings of the 7th ACM CCS International Workshop on Managing Insider Security Threats*. Association for Computing Machinery, 2015.

[43] Memcached Community. memcached. https://memcached.org, accessed on October 2021.

[44] Intel Corportation. Intel trusted execution techonology–software development guide, revision 017.0, 2008.

[45] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptol. ePrint Arch.*, 2016.

[46] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security '16)*, 2016.

[47] Jordi Cucurull and Sandra Guasch. Virtual TPM for a secure cloud: fallacy or reality? *RECSI 2014*, 2014.

[48] Rongzhen Cui, Lianying Zhao, and David Lie. Emilia: Catching iago in legacy code. In *28th Annual Network and Distributed System Security Symposium (NDSS '21)*, 2021.

[49] Natasha Dailey. The hackers that attacked a major US oil pipeline say it was only for money — here's what to know about DarkSide. https://www.businessinsider.com/pipeline-cyber-attack-darkside-hacker-group-shutdown-ransomware-money-politics-oil-2021-5?op=1&r=US&IR=T, accessed on May 2021.

[50] Dan Tarnovsky. DEF CON 20 - Attacking TPM Part 2 A Look at the ST19WP18 TPM Device. https://www.youtube.com/watch?v=Bp26rPw90Dc, accessed on October 2021.

[51] Janis Danisevskis, Michael Peter, Jan Nordholz, Matthias Petschick, and Julian Vetter. Graphical user interface for virtualized mobile handsets. *IEEE S&P MoST*, 2015.

[52] Ivan De Oliveira Nunes, Xuhua Ding, and Gene Tsudik. On the root of trust identification problem. In *Proceedings of the 20th International Conference on Information Processing in Sensor Networks (Co-Located with CPS-IoT Week 2021)*, 2021.

[53] Aritra Dhar, Ivan Puddu, Kari Kostiainen, and Srdjan Capkun. Proximitee: Hardened sgx attestation by proximity verification. In *Proceedings of the Tenth ACM Conference on Data and Application Security and Privacy*, CODASPY '20, 2020.

[54] T. Dierks and E. Rescorla. The Transport Layer Security Protocol Version 1.2. https://tools.ietf.org/html/rfc5246, accessed on October 2021.

[55] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. 1976.

[56] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 1983.

[57] Jon Emont, Laura Stevens, and Robert McMillan. Amazon Investigates Employees Leaking Data for Bribes. https://www.wsj.com/articles/amazon-investigates-employees-leaking-data-for-bribes-1537106401, accessed on October 2021.

[58] Eperi. Top Tier Bank and Confidential Computing. https://www.intel.com/content/www/us/en/customer-spotlight/stories/eperi-sgx-customer-story.html, accessed on October 2021.

[59] Gregor Erbach and Jack O'Shea. Cybersecurity of critical energy infrastructure. *European Parliamentary Research Service*, October 2019.

[60] Spyridon Bakas et al. Identifying the best machine learning algorithms for brain tumor segmentation, progression assessment, and overall survival prediction in the brats challenge, 2019.

[61] European Central Bank. Cyber resilience oversight expectations for financial market infrastructures – cyber resilience oversight expectations, 2018.

[62] European Commission. Proposal for a directive of the European Parliament and of the council on the resilience of critical entities, 2020.

[63] European Parliament. Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec, 2016.

[64] European Union. Horizon Europe Strategic Plan (2021 - 2024). Technical report, European Commission, 2021.

[65] F5, Inc. NGINX. https://www.nginx.com, accessed on October 2021.

[66] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine, 2000.

[67] Russell A Fink, Alan T Sherman, Alexander O Mitchell, and David C Challener. Catching the cuckoo: Verifying TPM proximity using a quote timing side-channel. In *International Conference on Trust and Trustworthy Computing*. Springer, 2011.

[68] Django Software Foundation. Django. https://www.djangoproject.com, accessed on October 2021.

[69] The Linux Foundation. The Update Framework Project. https://theupdateframework.github.io, accessed on October 2021.

[70] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model Inversion Attacks That Exploit Confidence Information and Basic Countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, 2015.

[71] Inc. Free Software Foundation. Tar - GNU Project - Free Software Foundation. https://www.gnu.org/software/tar/, accessed on October 2021.

[72] Paul W. Frields. Infrastructure report, 2008-08-22 UTC 1200. https://www.redhat.com/archives/fedora-announce-list/2008-August/msg00012.html, accessed on October 2021.

[73] Eimear Gallery and Chris J. Mitchell. Trusted Computing: Security and Applications. *Cryptologia*, 2009.

[74] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, 2003.

[75] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time Protection: The Missing OS Abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*, 2019.

[76] Gematik GmbH. Systemspezifisches Konzept E-Rezept. https://fachportal.gematik.de/fachportal-import/files/gemSysL_eRp_V1.1.0.pdf, accessed on October 2021.

[77] Gematik GmbH. Systemspezifisches Konzept ePA. https://www.vesta-gematik.de/standard/formhandler/324/gemSysL_ePA_V1_3_0.pdf, accessed on October 2021.

[78] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, 2009.

[79] Aleksander Gieysztor. *Mitologia Słowian*. Wydawnictwo Uniwersytetu Warszawskiego, 2006.

[80] Virgil Gligor and Maverick Woo. Establishing Software Root of Trust Unconditionally. In *Network and Distributed Systems Security (NDSS '19)*, 2019.

[81] GlobalPlatform Inc. The trusted execution environment: Delivering enhanced security at a lower cost to the mobile market. White paper, GlobalPlatform Inc., 2011.

[82] Johannes Goetzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Mueller. Cache Attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec '17)*, 2017.

[83] Kenneth Goldman and Stefan Berger. TPM Main Part 3 – IBM Commands. https://researcher.watson.ibm.com/researcher/files/us-kgoldman/mainP3IBMCommandsrev36.pdf, accessed on October 2021.

[84] Kenneth Goldman, Ronald Perez, and Reiner Sailer. Linking remote attestation to secure tunnel endpoints. In *Proceedings of the First ACM Workshop on Scalable Trusted Computing*, STC '06, 2006.

[85] James C Gordon. Microsoft Azure Confidential Computing with Intel SGX, accessed on October 2021.

[86] S. Govindavajhala and A.W. Appel. Using memory errors to attack a virtual machine. In *Proceedings of the 2003 Symposium on Security and Privacy, (S&P 2003)*, 2003.

[87] James Greene. Intel trusted execution technology: Hardware-based technology for enhancing server platform security. *Intel Corporation*, 2010.

[88] Franz Gregor, Wojciech Ozga, Sebastien Vaucher, Rafael Pires, Do Le Quoc, Sergei Arnautov, Andre Martin, Valerio Schiavoni, Pascal Felber, and Christof Fetzer. Trust management as a service: Enabling trusted execution in the face of byzantine stakeholders. In *The 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '20)*, 2020.

[89] Trusted Computing Group. TCG Infrastructure Working Group Architecture Part ii - Integrity Management, Specification Version 1.0, Revision 1.0, 2006.

[90] Trusted Computing Group. TPM Library Specification, family "2.0", level 00, revision 01.38. In *TCG Resources, TPM 2.0 Library*, 2016.

[91] Trusted Computing Group. TCG Trusted Attestation Protocol (TAP) Information Model for TPM Families 1.2 and 2.0 and DICE Family 1.0. Version 1.0, Revision 0.36, 2019.

[92] Trusted Computing Group. TPM Library Part 1: Architecture, Family "2.0", Level 00, Revision 01.38. http://www.trustedcomputinggroup.org/resources/tpm_library_specification, accessed on October 2021.

[93] Trusted Computing Group. Trusted Computing. https://trustedcomputinggroup.org/trusted-computing/, accessed on October 2021.

[94] Karan Grover, Shruti Tople, Shweta Shinde, Ranjita Bhagwan, and Ramachandran Ramjee. Privado: Practical and Secure DNN Inference with Enclaves. *arXiv preprint arXiv:1810.00602*, 2018.

[95] Le Guan, Jingqiang Lin, Bo Luo, Jiwu Jing, and Jing Wang. Protecting private keys against memory disclosure attacks using hardware transactional memory. In *2015 IEEE Symposium on Security and Privacy*, 2015.

[96] Marco Guarnieri, Boris Köpf, Jose F Morales, Jan Reineke, and Andres Sanchez. SPECTECTOR: Principled detection of speculative information flows. In *2020 IEEE Symposium on Security and Privacy (S&P '20)*. IEEE, 2020.

[97] Shay Gueron. Memory Encryption for General-Purpose Processors. *IEEE Security and Privacy*, 2016.

[98] Christian Göttel, Rafael Pires, Isabelly Rocha, Sébastien Vaucher, Pascal Felber, Marcelo Pasin, and Valerio Schiavoni. Security, Performance and Energy Trade-offs of Hardware-assisted Memory Protection Mechanisms. *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, 2018.

[99] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic Remote Attestation – A Virtual Machine directed approach to Trusted Computing. In *3rd Virtual Machine Research and Technology Symposium*, 2004.

[100] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Proceedings of the 17th USENIX Security Symposium (USENIX Security'08)*, 2008.

[101] Serge Hallyn, Dmitry Kasatkin, David Safford, Reiner Sailer, and Mimi Zohar. Linux Integrity Measurement Architecture (IMA) - IMA appraisal. https://sourceforge.net/p/linux-ima/wiki/Home/#ima-appraisal, accessed on October 2021.

[102] James Hamilton. AWS Nitro System. https://perspectives.mvdirona.com/2019/02/aws-nitro-system/, accessed on October 2021.

[103] Seunghun Han, Wook Shin, Jun-Hyeok Park, and HyoungChun Kim. A bad dream: Subverting trusted platform module while you are sleeping. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[104] Hex Five Security, Inc. MultiZone Hex Five Security. https://hex-five.com, accessed on October 2021.

[105] Guerney DH Hunt, Ramachandra Pai, Michael V Le, Hani Jamjoom, Sukadev Bhattiprolu, Rick Boivie, Laurent Dufour, Brad Frey, Mohit Kapur, Kenneth A Goldman, et al. Confidential computing for openpower. In *Proceedings of the 16th European Conference on Computer Systems (EuroSys '21)*, 2021.

[106] Tyler Hunt, Zhipeng Jia, Vance Miller, Ariel Szekely, Yige Hu, Christopher J. Rossbach, and Emmett Witchel. Telekine: Secure Computing with Cloud GPUs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, 2020.

[107] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving Machine Learning as a Service. *arXiv preprint arXiv:1803.05961*, 2018.

[108] Hyper. Hyper. https://hyper.rs, accessed on October 2021.

[109] IBM. IBM CEX7S / 4769 PCIe Cryptographic Coprocessor (HSM). *IBM 4769 Data Sheet*, 2019.

[110] IBM. IBM Security QRadar. Intelligent security analytics for insight into your most critical threats. https://www.ibm.com/security/security-intelligence/qradar, accessed on October 2021.

[111] IBM. IBM TPM Attestation Client Server. https://sourceforge.net/projects/ibmtpm20acs/, accessed on October 2021.

[112] IBM. Introducing IBM Secure Execution for Linux. https://www.ibm.com/support/knowledgecenter/linuxonibm/com.ibm.linux.z.lxse/lxse_t_secureexecution.html, accessed on October 2021.

[113] IBM X-Force Incident Response and Intelligence Services (IRIS). X-force threat intelligence index. In *IBM Security report*, 2020.

[114] IBM X-Force Incident Response and Intelligence Services (IRIS). X-force threat intelligence index. In *IBM Security report*, 2021.

[115] IEEE and The Open Group. The Open Group Base Specifications Issue 7, 2018 edition, IEEE std 1003.1-2017. https://pubs.opengroup.org/onlinepubs/9699919799/utilities/pax.html#tag_20_92_13_03, accessed on October 2021.

[116] Advanced Micro Devices Inc. AMD Secure Encrypted Virtualization API Version 0.22. *Technical Preview*, 2019.

[117] Chef Software Inc. Chef. https://www.chef.io/chef/, accessed on October 2021.

[118] Free Software Foundation Inc. Basic Tar Format. https://www.gnu.org/software/tar/manual/html_node/Standard.html, accessed on October 2021.

[119] Puppet Inc. Puppet - server automation framework and application. https://puppet.com, accessed on October 2021.

[120] Information Technology Laboratory. Security requirements for cryptographic modules. Federal Information Processing Standards Publication, 2001.

[121] Intel. Strengthening Security with Intel Platform Trust Technology. In *Intel Whitepaper*, 2014.

[122] Intel. Memory Encryption Technologies Specification Rev: 1.2. Intel Architecture, Intel, 2019.

[123] Intel. Intel Trust Domain Extensions. *Intel White Paper*, 2020.

[124] Intel. Intel OpenAttestation project. https://github.com/OpenAttestation/OpenAttestation, accessed on October 2021.

[125] Intel. Intel Security Libraries for Data Center. https://01.org/intel-secl, accessed on October 2021.

[126] Intel. Resources and Response to Side Channel L1TF. https://www.intel.com/content/www/us/en/architecture-and-technology/l1tf.html, accessed on October 2021.

[127] Intel. Trusted Boot (tboot). https://sourceforge.net/projects/tboot/, accessed on October 2021.

[128] Intel and National Security Agency. Intel Open Cloud Intergrity Technology. https://01.org/opencit, accessed on October 2021.

[129] Intel Corporation. Intel SGX: Intel EPID Provisioning and Attestation Services. https://software.intel.com/content/www/us/en/develop/download/intel-sgx-intel-epid-provisioning-and-attestation-services.html, accessed on October 2021.

[130] Trent Jaeger, Reiner Sailer, and Umesh Shankar. Prima: Policy-reduced integrity measurement architecture. In *Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies*, SACMAT '06, 2006.

[131] Insu Jang, Adrian Tang, Taehoon Kim, Simha Sethumadhavan, and Jaehyuk Huh. Heterogeneous Isolated Execution for Commodity GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*, 2019.

[132] Ramya Jayaram Masti, Claudio Marforio, and Srdjan Capkun. An architecture for concurrent execution of secure environments in clouds. In *Proceedings of the 2013 ACM workshop on Cloud computing security workshop*, 2013.

[133] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. Intel software guard extensions: EPID provisioning and attestation services. *White Paper*, 2016.

[134] Joseph Birr-Pixton. rustls. https://github.com/ctz/rustls, accessed on October 2021.

[135] Ashlesha Joshi, Samuel T. King, George W. Dunlap, and Peter M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, 2005.

[136] Rob Joyce. Disrupting Nation State Hackers. *USENIX Enigma'16*, 2016.

[137] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *Proceedings of the 27th USENIX Conference on Security Symposium (USENIX Security)*, 2018.

[138] Georgios A Kaissis, Marcus R Makowski, Daniel Rückert, and Rickmer F Braren. Secure, privacy-preserving and federated machine learning in medical imaging. *Nature Machine Intelligence*, 2020.

[139] David Kaplan. Protecting VM register State with SEV-ES. AMD White Paper, AMD, 2017.

[140] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. Amd white paper, AMD, 2016.

[141] Karnati. Data-in-use protection on IBM Cloud using Intel SGX. https://www.ibm.com/blogs/bluemix/2018/05/data-use-pro-tection-ibm-cloud-using-intel-sgx/, accessed on October 2021.

[142] Bernhard Kauer. OSLO: Improving the security of Trusted Computing. *USENIX*, 2007.

[143] Mustakimur Rahman Khandaker, Wenqing Liu, Abu Naser, Zhi Wang, and Jie Yang. Origin-sensitive control flow integrity. In *28th USENIX Security Symposium (USENIX Security '19)*, 2019.

[144] Sven Kiljan, Koen Simoens, Danny De Cock, Marko Van Eekelen, and Harald Vranken. A survey of authentication and communications security in online banking. *ACM Computing Surveys (CSUR)*, 49(4), 2016.

[145] S.T. King and P.M. Chen. SubVirt: implementing malware with virtual machines. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*, 2006.

[146] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[147] Avi Kivity, Yaniv Kamay, and Dor Laor. kvm: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium, Volume One*, 2007.

[148] Gerwin Klein, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, and Rafal Kolanski. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*, Big Sky, Montana, USA, 2009.

[149] 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.

[150] Jeffrey Knockel and Jedidiah R. Crandall. Protecting Free and Open Communications on the Internet Against Man-in-the-Middle Attacks on Third-Party Software: We're FOCI'd. In *Proceedings of the 2nd USENIX Workshop on Free and Open Communications on the Internet (FOCI '12)*. USENIX, 2012.

[151] 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)*, 2019.

[152] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, 1996.

[153] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '99, 1999.

[154] Philipp Koppe, Benjamin Kollenda, Marc Fyrbiak, Christian Kison, Robert Gawlik, Christof Paar, and Thorsten Holz. Reverse engineering x86 processor microcode. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[155] Kari Kostiainen, Aritra Dhar, and Srdjan Capkun. Dedicated Security Chips in the Age of Secure Enclaves. *IEEE Security and Privacy*, 2020.

[156] Robert Krahn, Donald Dragoti, Franz Gregor, Do Le Quoc, Valerio Schiavoni, Pascal Felber, Clenimar Souza, Andrey Brito, and Christof Fetzer. TEEMon: A continuous performance monitoring framework for TEEs. In *Proceedings of the 21th International Middleware Conference (Middleware)*, 2020.

[157] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[158] Kubernetes. Kubernetes website. https://kubernetes.io/, accessed on October 2021.

[159] Ambuj Kumar, Anand Kashyap, Vinay Phegade, and Jesse Schrater. Self-Defending Key Management Service with Intel SGX. *Fortranix Whitepaper*, accessed on October 2021.

[160] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. CrypTFlow: Secure TensorFlow Inference. In *IEEE Symposium on Security and Privacy (S&P '20)*, 2020.

[161] Trishank Karthik Kuppusamy, Vladimir Diaz, and Justin Cappos. Mercury: Bandwidth-Effective Prevention of Rollback Attacks against Community Repositories. In *Proceedings of the 2017 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC '17)*. USENIX Association, 2017.

[162] Trishank Karthik Kuppusamy, Santiago Torres-Arias, Vladimir Diaz, and Justin Cappos. Diplomat: Using Delegations to Protect Community Repositories. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation (NSDI '16)*, 2016.

[163] Klaus Kursawe, Dries Schellekens, and Bart Preneel. Analyzing trusted platform communication. In *ECRYPT Workshop, CRASH-CRyptographic Advances in Secure Hardware*, 2005.

[164] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical Cache Attacks from the Network. In *2020 IEEE Symposium on Security and Privacy (S&P '20)*, 2020.

[165] Matthias Lange and Steffen Liebergeld. Crossover: secure and usable user interface for mobile devices with multiple isolated os personalities. In *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013.

[166] Hagen Lauer, Amin Sakzad, Carsten Rudolph, and Surya Nepal. Bootstrapping Trust in a "Trusted" Virtualized Platform. In *Proceedings of the 1st ACM Workshop on Workshop on Cyber-Security Arms Race (CYSARM '19)*, 2019.

[167] Do Le Quoc, Franz Gregor, Sergei Arnautov, Roland Kunkeland, Pramod Bhatotia, and Christof Fetzer. secureTF: A Secure TensorFlow Framework. In *Proceedings of the 21th International Middleware Conference (Middleware)*, 2020.

[168] Do Le Quoc, Franz Gregor, Jatinder Singh, and Christof Fetzer. SGX-PySpark: Secure Distributed Data Analytics. In *The World Wide Web Conference (WWW '19)*, 2019.

[169] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*, 2020.

[170] Jun Li, P. L. Reiher, and G. J. Popek. Resilient Self-Organizing Overlay Networks for Security Update Delivery. *IEEE Journal on Selected Areas in Communications*, 22(1), 2006.

[171] Christopher Liebchen. Advancing Memory-corruption Attacks and Defenses. *System Security Lab Fachbereich fur Informatik Technische Universitaet Darmstadt*, 2018.

[172] ARM Limited. Building a Secure System using TrustZone Technology. White paper, accessed on October 2021.

[173] Alpine Linux. Alpine Linux community repository. http://dl-cdn.alpinelinux.org/alpine/edge/community/, accessed on October 2021.

[174] Alpine Linux. Alpine Linux main repository. http://dl-cdn.alpinelinux.org/alpine/edge/main/, accessed on October 2021.

[175] Alpine Linux. Alpine Linux package management. https://wiki.alpinelinux.org/wiki/Alpine_Linux_package_management, accessed on October 2021.

[176] Debian Linux. Debian Linux: Debian package management. https://www.debian.org/doc/manuals/debian-reference/ch02.en.html, accessed on October 2021.

[177] Gentoo Linux. Gentoo Linux: Portage build system. https://wiki.gentoo.org/wiki/Portage, accessed on October 2021.

[178] Theo Markettos, Colin Rothwell, Brett F Gutstein, Allison Pearce, Peter G Neumann, Simon Moore, and Robert Watson. Thunderclap: Exploring vulnerabilities in operating system iommu protection via dma from untrustworthy peripherals. In *Network and Distributed System Security Symposium*, 2019.

[179] Sinisa Matetic, David Sommer, Mansoor Ahmed, Arthur Gervais, Kari Kostiainen, Aritra Dhar, Ari Juels, and Srdjan Capkun. ROTE: Rollback Protection for Trusted Execution. *26th USENIX Security Symposium (USENIX Security '17)*, 2017.

[180] Nicholas D Matsakis and Felix S Klock. The Rust language. *ACM SIGAda Ada Letters*, 2014.

[181] Matthew Garrett. dpkg patch. https://gitlab.com/mjg59/dpkg/-/commits/master, accessed on October 2021.

[182] J.M. McCune, A. Perrig, and M.K. Reiter. Seeing-is-believing: using camera phones for human-verifiable authentication. In *2005 IEEE Symposium on Security and Privacy (S&P'05)*, 2005.

[183] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. TrustVisor: Efficient TCB reduction and attestation. In *2010 IEEE Symposium on Security and Privacy*, 2010.

[184] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, 2008.

[185] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *HASP isca*, 2013.

[186] Slashdot Media. phpMyAdmin corrupted copy on Korean mirror server. https://sourceforge.net/blog/phpmyadmin-back-door/, 2012.

[187] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A Cryptographic Inference Service for Neural Networks. In *29th USENIX Security Symposium (USENIX Security '20)*, 2020.

[188] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy (S&P '17)*, 2017.

[189] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P '20)*, 2020.

[190] muslc. musl libc. https://musl.libc.org, accessed on October 2021.

[191] Eugene D Myers. Using the intel stm for protected execution. https://www.platformsecuritysummit.com/2018/speaker/myers/STMPE2Intelv84a.pdf, last accessed on July 2021.

[192] Lucien KL Ng, Sherman SM Chow, Anna PY Woo, Donald PH Wong, and Yongjun Zhao. Goten: GPU-Outsourcing Trusted Execution of Neural Network Training and Prediction. *35th AAAI Conference on Artificial Intelligence*, 2019.

[193] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. CHAINIAC: Proactive software-update transparency via collectively signed skipchains and verified builds. In *26th USENIX Security Symposium (USENIX Security '17)*, 2017.

[194] NIST. CVE-2019-5021. https://nvd.nist.gov/vuln/detail/CVE-2019-5021, accessed on October 2021.

[195] NIST. The Heartbleed Bug: CVE-2014-0160. https://nvd.nist.gov/vuln/detail/CVE-2014-0160, accessed on October 2021.

[196] Talal H. Noor, Quan Z. Sheng, Sherali Zeadally, and Jian Yu. Trust Management of Services in Cloud Environments: Obstacles and Solutions. *ACM Comput. Surv.*, 2013.

[197] OASIS. PKCS#11 specification. http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.html, accessed on October 2021.

[198] National Institute of Standards and Technology (NIST). National Software Reference Library (NSRL). https://www.nist.gov/itl/ssd/software-quality-group/national-software-reference-library-nsrl/about-nsrl/nsrl-introduction, accessed on October 2021.

[199] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious Multi-Party Machine Learning on Trusted Processors. In *Proceedings of the 25th USENIX Conference on Security Symposium*, 2016.

[200] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *2018 Usenix Annual Technical Conference (USENIX ATC '18)*, 2018.

[201] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security '20)*, 2020.

[202] OpenStack community. OpenStack. https://www.openstack.org, accessed on October 2021.

[203] Wojciech Ozga, Rasha Faqeh, Do Le Quoc, Franz Gregor, Silvio Dragone, and Christof Fetzer. Chors: Hardening high-assurance security systems with trusted computing. *Proceedings of the 37th ACM Symposium On Applied Computing (SAC '22)*, 2022.

[204] Wojciech Ozga, Do Le Quoc, and Christof Fetzer. A practical approach for updating an integrity-enforced operating system. In *Proceedings of the 21st International Middleware Conference*, 2020.

[205] Bryan Parno. Bootstrapping Trust in a Trusted Platform. In *Proceedings of the 3rd Conference on Hot Topics in Security*, 2008.

[206] LLC PCI Security Standards Council. Requirements and Security Assessment Procedures, Version 3.2.1. *Payment Card Industry (PCI) Data Security Standard*, 2018.

[207] Mike Petullo. Encrypt your root filesystem. *Linux Journal*, 2005.

[208] Jonathan Poritz, Matthias Schunter, Els Van Herreweghen, and Michael Waidner. Property attestation — Scalable and privacy-friendly security assessment of peer computers. In *IBM Research Report, Computer Science RZ3548*, 2004.

[209] Dr Daniel Potts, Rene Bourquin, Leslie Andresen, Dr Gerwin Klein, and Gernot Heiser. Mathematically Verified Software Kernels: Raising the Bar for High Assurance Implementations. Article, General Dynamics, 2014.

[210] Bart Preneel. The state of cryptographic hash functions. In *Lectures on Data Security, Modern Cryptology in Theory and Practice, Summer School, Aarhus, Denmark, July 1998*, 1999.

[211] Emil Protalinski. TechSpot News. Google fired employees for breaching user privacy. https://www.techspot.com/news/40280-google-fired-employees-for-breaching-user-privacy.html, accessed on October 2021.

[212] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, et al. Evercrypt: A fast, verified, cross-platform cryptographic provider. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.

[213] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, Magnus Nystrom, David Robinson, Rob Spiger, Stefan Thom, and David Wooten. fTPM: A Software-only Implementation of a TPM Chip. *Proceedings of the 25th USENIX Security Symposium*, 2016.

[214] RedHat, Inc. Critical: openssh security update. https://access.redhat.com/errata/RHSA-2008:0855, accessed on October 2021.

[215] RedHat, Inc. OpenSCAP - Audit, Fix and be Merry. https://www.open-scap.org, accessed on October 2021.

[216] Redis Labs. NoSQL Redis and Memcache traffic generation and benchmarking tool. https://github.com/RedisLabs/memtier_benchmark, accessed on October 2021.

[217] Max Reitz. *Isolating Program Execution on L4Re/Fiasco.OC*. PhD thesis, TU Dresden, 2019.

[218] Reuters. Ex-Microsoft employee charged with leaking trade secrets to blogger. https://www.reuters.com/article/us-microsoft-tradesecret-idUSBREA2J07K20140320, accessed on October 2021.

[219] Reuters. Foreign Hackers Probe European Critical Infrastructure Networks: Sources. https://www.reuters.com/article/us-britain-cyber-idINKBN19V1C7, accessed on October 2021.

[220] Jordan Robertson and Michael Riley. The Big Hack: How China Used a Tiny Chip to Infiltrate U.S. Companies. *Bloomberg Businessweek*, 2018.

[221] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, 2015.

[222] Joanna Rutkowska. Introducing blue pill. *SyScan'06*, 2006.

[223] 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*, 2004.

[224] Reiner Sailer, Enriquillo Valdez, Trent Jaeger, Ronald Perez, Leendert Van Doorn, John Linwood Griffin, and Stefan Berger. shype: Secure hypervisor approach to trusted virtualized systems. In *IBM research report RC23511*, 2005.

[225] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert Van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *USENIX Security symposium*, 2004.

[226] Roberto Sassu. Infoflow LSM. In *Linux Security Summit'19*, 2019.

[227] Uday Savagaonkar, Nelly Porter, Nadim Taha, Benjamin Serebrin, and Neal Mueller. Titan in depth: Security in plaintext. *Google Cloud Identity and Security Blog*, 2017.

[228] 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.

[229] Nabil Schear, Patrick T. Cable, Thomas M. Moyer, Bryan Richard, and Robert Rudd. Bootstrapping and maintaining trust in the cloud. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC '16)*, 2016.

[230] Joshua Schiffman, Hayawardh Vijayakumar, and Trent Jaeger. Verifying System Integrity by Proxy. In *Trust and Trustworthy Computing*. Springer Berlin Heidelberg, 2012.

[231] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2017.

[232] Scontain UG. SCONE Docker curated images. https://hub.docker.com/u/sconecuratedimages, accessed on October 2021.

[233] Security Standards Council, LLC. Payment Card Industry (PCI) Data Security Standard. Requirements and Security Assessment Procedures. Version 3.2.1, 2018.

[234] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP '05)*, 2005.

[235] Jacob Shin, Bill Jacobs, Mark Scott-Nash, Julian Hammersley, Monty Wiseman, Rob Spiger, Dick Wilkins, Ralf Findeisen, David Challener, Dalvis Desselle, Steve Goodman, Gary Simpson, Kirk Brannock, Amy Nelson, Mark Piwonka, Conan Dailey, and Randy Springfield. TCG D-RTM Architecture, Document Version 1.0.0. *Trusted Computing Group*, 2013.

[236] Amber L Simpson, Michela Antonelli, Spyridon Bakas, Michel Bilello, Keyvan Farahani, Bram Van Ginneken, Annette Kopp-Schneider, Bennett A Landman, Geert Litjens, Bjoern Menze, et al. A large annotated medical image dataset for the development and evaluation of segmentation algorithms. *arXiv preprint arXiv:1902.09063*, 2019.

[237] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. Microscope: Enabling microarchitectural replay attacks. In *Proceedings of the 46th International Symposium on Computer Architecture*, ISCA '19, 2019.

[238] Evan R Sparks. A Security Assessment of Trusted Platform Modules. *Computer Science Technical Report TR2007-597*, 2007.

[239] Stefan Berger. [PATCH v2] Support for PAX extended header and Linux extended attributes. https://linux.debian.maint.dpkg.narkive.com/Jwr2kstj/patch-v2-support-for-pax-extended-header-and-linux-extended-attributes, accessed on October 2021.

[240] Andreas Steffen. StrongSwan an OpenSource IPsec implementation. https://www.strongswan.org, accessed on October 2021.

[241] Doug Stiles. The Hardware Security Behind Azure Sphere. *IEEE Micro*, 2019.

[242] Raoul Strackx and Frank Piessens. Ariadne: A Minimal Approach to State Continuity. *25th USENIX Security Symposium (USENIX Security '16)*, 2016.

[243] Frederic Stumpf and Claudia Eckert. Enhancing Trusted Platform Modules with Hardware-Based Virtualization Techniques. In *2008 Second International Conference on Emerging Security Information, Systems and Technologies*. IEEE, 2008.

[244] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Haining Wang. Trustice: Hardware-assisted isolated computing environments on mobile devices. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*, 2015.

[245] Sahil Suneja, Canturk Isci, Eyal de Lara, and Vasanth Bala. Exploring VM Introspection: Techniques and Trade-offs. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments - VEE '15*, Istanbul, Turkey, 2015.

[246] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018.

[247] Dina Temple-Raston. A 'worst nightmare' cyberattack: The untold story of the solarwinds hack. https://www.npr.org/2021/04/16/985439655/a-worst-nightmare-cyberattack-the-untold-story-of-the-solarwinds-hack, accessed on October 2021.

[248] Gil Tene et al. wrk2 HTTP benchmarking tool. https://github.com/giltene/wrk2, accessed on October 2021.

[249] The White House. Executive Order on Improving the Nation's Cybersecurity. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/, accessed on October 2021.

[250] The New York Times. Hack of Saudi Petrochemical Plant Was Coordinated From Russian Institute. https://www.nytimes.com/2018/10/23/us/politics/russian-hackers-saudi-chemical-plant.html, accessed on October 2021.

[251] The New York Times. Hackers Are Targeting Nuclear Facilities, Homeland Security Dept. and F.B.I. Say. https://www.nytimes.com/2017/07/06/technology/nuclear-plant-hack-report.html, accessed on October 2021.

[252] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium (USENIX Security '19)*, 2019.

[253] Florian Tramèr and Dan Boneh. Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware. *7th International Conference on Learning Representations (ICLR)*, 2019.

[254] Trusted Computing Group. TCG PC Client Specific Implementation Specification for Conventional BIOS, Specification Version 1.21, Revision 1.00, 2012.

[255] Trusted Computing Group. TCG PC Client Platform Firmware Profile Specification, Family 2.0, Level 00, Revision 1.04, 2019.

[256] Trusted Computing Group. Trusted Computing Group Website. https://trustedcomputinggroup.org, accessed on October 2021.

[257] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*, 2017.

[258] Scontain UG. SCONE Configuration and Attestation Service (CAS). https://sconedocs.github.io/CASOverview/, accessed on October 2021.

[259] Scontain UG. SCONE Rust cross-compilers. https://hub.docker.com/r/sconecuratedimages/rust, accessed on October 2021.

[260] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[261] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[262] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. SGAxe: How SGX fails in practice. https://sgaxeattack.com/, 2020.

[263] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: Secure Multi-Party Computation on Big Data. In *Proceedings of the 14th EuroSys Conference (EuroSys '19)*, 2019.

[264] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted Execution Environments on GPUs. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI '18)*, 2018.

[265] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V. In *Network and Distributed Systems Security (NDSS '19)*, 2019.

[266] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. NDA: Preventing Speculative Execution Attacks at Their Source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[267] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. *Technical report*, 2018.

[268] David Weston. Microsoft and partners design new device security requirements to protect against targeted firmware attacks. *Microsoft Security Blog*, 2019.

[269] Richard Wilkins and Brian Richardson. UEFI secure boot in modern computer security solutions. In *UEFI Forum*, 2013.

[270] Johannes Winter. Eavesdropping trusted platform module communication. In *Presented at 4th European Trusted Infrastructure Summer school, ETISS 2009*, 2009.

[271] Johannes Winter and Kurt Dietrich. A Hijacker's Guide to the LPC bus. In *Public Key Infrastructures, Services and Applications*. Springer Berlin Heidelberg, 2011.

[272] Johannes Winter and Kurt Dietrich. A hijacker's guide to communication interfaces of the trusted platform module. *Computers & Mathematics with Applications*, 2013.

[273] Rafal Wojtczuk and Joanna Rutkowska. Attacking Intel Trusted Execution Technology. In *Black Hat DC*, 2009.

[274] Rafal Wojtczuk and Joanna Rutkowska. Attacking Intel TXT via SINIT code execution hijacking. https://invisiblethingslab.com/resources/2011/Attacking_Intel_TXT_via_SINIT_hijacking.pdf, accessed on October 2021.

[275] Ruan Xiaoyu. *Platform Embedded Security Technology Revealed. Safeguarding the Future of Computing with Intel Embedded Security and Management Engine*. Apress Open, 2014.

[276] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do Not Blame Users for Misconfigurations. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*, 2013.

[277] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (S&P '15)*, 2015.

[278] Andrew C Yao. Protocols for secure computations. In *23rd IEEE Annual Symposium on Foundations of Computer Science (SFCS 1982)*, 1982.

[279] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

[280] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. RFC 4254 (Proposed Standard). Updated by RFC 6668. https://tools.ietf.org/html/rfc6668, 2006.

[281] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. The fuzzing book, 2019.

[282] Lianying Zhao and David Lie. Is hardware more secure than software? *IEEE Security & Privacy*, 2020.

[283] Lianying Zhao and Mohammad Mannan. Hypnoguard: Protecting secrets across sleep-wake cycles. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*, 2016.

[284] Lianying Zhao and Mohammad Mannan. Tee-aided write protection against privileged data tampering. In *26th Annual Network and Distributed System Security Symposium (NDSS '19)*, 2019.

[285] Oliver Zheng, Jason Poon, and Konstantin Beznosov. Application-based TCP hijacking. In *Proceedings of the Second European Workshop on System Security - EUROSEC '09*. ACM Press, 2009.

[286] Lei Zhou, Fengwei Zhang, Jinghui Liao, Zhengyu Ning, Jidong Xiao, Kevin Leach, Westley Weimer, and Guojun Wang. KShot: Live Kernel Patching with SMM and SGX. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '20)*, 2020.

[287] Jean-Karim Zinzindohoue, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

Notices

Trademark

IBM is a trademark or registered trademark of International Business Machines Corporation, in the United States and/or other countries. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on ibm.com/trademark.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

The registered trademark Linux is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.