Enhancing Public Cloud Attestation with the SVSM-vTPM and Keylime

by
De Moerloose Jelle
School of Electrical Engineering and Computer Science,
The University of Queensland.

Submitted for the degree of
Master of Engineering
in Information Technology
JUNE & 2025.

6 Jane Street , West End, QLD 4101
Tel. +32471293979

June 7, 2025

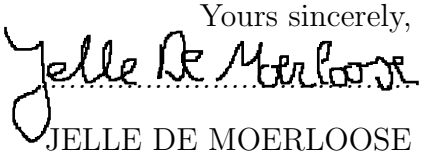
Prof Michael Bruenig
Head of School
School of Electrical Engineering and Computer Science
The University of Queensland
St Lucia, Q 4072

Dear Professor Bruenig,

In accordance with the requirements of the degree of Master of Engineering in the division of Electrical Engineering, Electrical and Biomedical Engineering, Electrical and Computer Engineering, Software Engineering, Mechatronic Engineering, I present the following thesis entitled "Enhancing Public Cloud Attestation with the SVSM-vTPM and Keylime". This work was performed under the supervision of Mr Omar Jarkas and Prof. Ryan Ko.

I declare that the work submitted in this thesis is my own, except as acknowledged in the text and footnotes, and has not been previously submitted for a degree at The University of Queensland or any other institution.

Yours sincerely,



JELLE DE MOERLOOSE

To my family, for their unwavering support.

Acknowledgments

I would like to express my deepest gratitude to my primary supervisor, Mr. Omar Jarkas, for his support, frequent meetings, and insightful guidance throughout this project. His encouragement and ability to challenge and reassure my ideas helped me approach my goals in a systematic and high-level manner. I have learned a great deal under his mentorship, and his dedication has been instrumental to the success of this thesis.

I would also like to thank Professor Ryan Ko for his role as co-supervisor and for providing valuable feedback during evaluations and key moments in the project. His input helped shape the final direction of this work.

I am sincerely thankful to the University of Queensland and my home institution, Ghent University, for offering me the opportunity to take part in this unique one-semester exchange thesis. It has been a meaningful, enriching, and truly extraordinary experience.

Finally, I wish to thank my family and my girlfriend, Iefke Rokegem, for their unwavering support during my exchange semester. Their encouragement and belief in me gave me the strength to stay committed and work hard on this project.

Acknowledgement of the Use of Generative AI and Machine Translation

Part 1. Acknowledgement

Yes. Generative AI and/or machine translation (MT) tools have been used to generate material in this document.

Part 2. Details of Use

Tool	**Use**	**Prompt(s)**	**Section/Page**
Grammerly	Automatically detecting spelling and grammar errors, and suggesting improvements to sentence structure.	No prompt, just passive suggestions and error flags within the text editor.	All sections
ChatGPT 4o	Proofreading and improving the structure and flow of some paragraphs.	Improve this paragraph for clarity.	All sections
ChatGPT 4o	Brainstorming on how to structure the implementation chapter.	For my implementation, I had a failed approach initially, how can I integrate this into this chapter.	Chapter 4

Abstract

The growing adoption of cloud computing, driven by scalability and cost-efficiency, raises concerns about data confidentiality and workload integrity. Confidential computing aims to address this by leveraging hardware-based remote attestation protocols to eliminate reliance on the cloud provider. Public cloud providers have introduced confidential computing features such as virtual TPMs (vTPMs) and attestation services. However, these often fall short because they are closed-source, managed by the hypervisor, or not backed by hardware. One promising solution, the SVSM-vTPM, proposes a stateless vTPM deployed inside a Trusted Execution Environment (TEE) using AMD SEV-SNP and VMPLs. While it eliminates trust in the provider, its feasibility on public cloud platforms remains unexplored. Key to its design is the use of Keylime for independent attestation, yet Keylime's complexity and limited documentation pose deployment challenges. This thesis investigates the feasibility of deploying the SVSM-vTPM on AWS, Azure, and GCP, evaluating each against high-level requirements derived from the original design and Proof-of-Concept. The analysis found that none of the platforms currently support the necessary capabilities, rendering direct deployment infeasible. Nonetheless, the design remained relevant, serving as a benchmark in a comparative analysis of cloud-provided vTPMs, which identified Azure's as the closest match. To pursue independent attestation within current limitations, the thesis pivots to deploying Keylime on Azure as a practical alternative. The implementation followed an incremental approach: beginning with a basic single-VM setup, then extending to a full multi-VM architecture setup using Azure's vTPM. To address Keylime's complexity and poor documentation, the project developed supporting tools including Ansible automation, infrastructure-as-code via Pulumi, and detailed configuration guides. While full hardware-backed attestation was not achieved, the resulting system enables reproducible attestation on Azure and lowers the barrier for future adopters of Keylime.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter introduces the project's field and its relevance, followed by a high-level overview of the problem statement. Next, the project's goals are defined, highlighting their potential impact on the field. Finally, the structure and content of the remaining chapters will be outlined, serving as a roadmap for the rest of the thesis. This chapter may introduce new technologies and difficult concepts, but these will be explained in Chapter 2, where relevant literature is reviewed.

1.1 Context and Relevance

Today, more and more companies and individuals are turning to the cloud to store data, execute their workloads, or host their services. Cloud providers, like AWS, Microsoft, and Google, rent out their existing hardware fleet to the public. This service model is called **Infrastructure-as-a-Service (IaaS)** and it provides many benefits to the consumer, aiding its widespread adoption. Most providers only charge based on usage, allowing a customer to use services while avoiding unnecessary investments or maintenance costs. Additionally, it makes it easier to scale workloads depending on their demand or spread them geographically [1].

With the new wave of AI advancements, particularly in deep learning and large-scale model training, the role of cloud computing has become more crucial than ever. Cloud computing plays a key role by providing the scalable infrastructure necessary to train complex AI models, making it an even more interesting option [2].

As the adoption of cloud computing surges, so do the concerns regarding data and execution security. Although these companies claim they offer great security, breaches still frequently occur, such as the Facebook data leak in 2019 [3]. Until recently, companies working with highly sensitive data needed to trust the cloud provider to

ensure confidentiality and integrity of the data, which is not ideal.

An emerging field, called confidential computing, tries to address this problem by removing the need for trust in the cloud provider. This is achieved by leveraging trusted hardware and attestation protocols. Due to data privacy regulations like the **General Data Protection Regulation (GDPR)** [4], **Health Insurance Portability and Accountability Act (HIPAA)** [5] and many more, companies and other institutes (hospitals, universities, governments,..) are forced to turn to confidential computing as it provides these strong security guarantees. Additionally, the growing adoption of AI technologies in privacy-sensitive environments, such as hospitals, is further driving the demand for confidential computing. As a result, the global confidential computing market is expected to be worth 59.4 billion USD by 2028, which is a 62-fold increase compared to 2023 [6].

Despite its promise, confidential computing is still in its early stages, facing challenges related to cost, implementation complexity, and security limitations. Current designs often have vulnerabilities or fail to meet all the security requirements for confidential computing. This project examines one promising design, the SVSM-vTPM, introduced by Narayanan et al. in [7], that could fully remove trust in the cloud provider.

1.2 Problem Statement

In recent years, popular cloud platforms such as Amazon Web Services, Azure, and Google Cloud have been working to provide cloud consumers with confidential computing solutions. These initiatives aim to strengthen the traditionally weak trust guarantees by shifting toward hardware-based solutions that offer a strong foundation to build trust on. This hardware root of trust provides stronger trust guarantees since it is much harder to subvert than software-based mechanisms [8, 9].

One prominent example of such a hardware-based solution is the **Confidential Virtual Machine (CVM)**, which enables consumers to securely deploy applications that handle sensitive data in fields like finance, healthcare, and AI. CVMs are built on **Trusted Execution Environments (TEEs)**, which ensure data confidentiality during execution.

A key component in this security model is the **virtual Trusted Platform Module (vTPM)**, which is used to ensure the integrity of all binaries being executed or dynamically loaded onto a virtual machine (VM). The cloud consumer, can use this vTPM to cryptographically verify that a VM is in a trusted state, in a process called **remote attestation**.

The problem is that most cloud providers offer vTPMs that are managed by the hypervisor, closed-sourced, and not even chained to a hardware root of trust [7]. Consequently, they can be tampered with by the hypervisor, be implemented insecurely, and contain backdoors. Additionally, public cloud consumers often cannot perform remote attestation themselves and must rely on the provider to handle this process [10]. These two issues are very problematic, as they reintroduce the need for trust in the cloud provider, which is what these solutions aimed to prevent.

To address these concerns, some solutions place the vTPM inside a TEE, isolating it from the cloud provider and reducing the risk of tampering. The paper by Narayanan et al. [7] introduces such a TEE-based vTPM, called the SVSM-vTPM. It has a simpler and more secure design than other TEE-based vTPMs and does not maintain state across reboots, eliminating risks associated with persistent state management. Furthermore, this design uses Keylime [11], an open-source framework that enables remote attestation to be performed independently of the cloud provider.

While the design of the SVSM-vTPM by Narayanan et al. [7] has promising security and performance properties, the paper lacks comprehensive implementation details and has only been implemented on an experimental bare-metal cloud environment as a Proof-of-Concept, available on Github [12]. This raises the question of whether the design is feasible on current public cloud platforms, which are generally more restrictive than research-based platforms.

Moreover, the deployment of Keylime for secure remote attestation presents additional challenges. Although Keylime tries to simplify the attestation process, it is a complex multicomponent software with very limited and incomplete documentation [11, 13]. Consequently, this can hinder its adoption by cloud consumers, especially those with limited experience in confidential computing.

1.3 Project Goals

This project aims to address the gaps discussed in Section 1.2 of the work by Narayanan et al. [7] and Keylime to maximize the security guarantees of consumers of public cloud platforms. To provide a clearer understanding of the project's goals, the following goals are outlined.

- **Assessing and documenting the applicability of SVSM-vTPM in modern confidential cloud computing offerings.** This will be achieved by thoroughly researching public cloud platforms' offerings and their respective consumer capabilities, and evaluating them against requirements deduced from the SVSM-vTPM design. Furthermore, the design will be implemented

incrementally on the platform that meets the criteria, and the implementation challenges and insights will be documented in great detail. Through this process, the study aims to derive industry-relevant insights that improve the development of secure remote attestation mechanisms within existing cloud infrastructures.

- **Deploying secure independent remote attestation with Keylime and enhancing its documentation.** This objective involves a comprehensive exploration of the Keylime framework to understand its typical workflows and functionalities. The goal is to eventually perform independent attestation on public cloud infrastructure, replacing the need for their provided closed-sourced attestation services. Given the current limitations and underdevelopment of Keylime's official documentation, this endeavor aims to provide comprehensive documentation to assist others in implementing their independent remote attestation solutions. These efforts will directly support the implementation and attestation of the SVSM-vTPM design, described in the first goal.

With these goals, this project embarks on a complex journey through the evolving field of confidential computing. The field is characterized by rapidly evolving technologies, limited documentation, and complex concepts that require a deep understanding. By navigating these complexities, this work aims to provide valuable insights into current public cloud offerings and the deployment of remote attestation mechanisms. Each encountered obstacle will be carefully documented, offering a comprehensive guide for newcomers to the field.

1.4 Project Overview

This Section will shortly give an idea of how this thesis is structured and what to expect from each of the following Chapters. It will act as a guideline to help better understand the big picture.

Chapter 2 summarizes the necessary background and related work based on a range of high-quality papers and sources. It introduces key concepts in confidential computing, illustrates them with relevant examples, and concludes with an overview of SVSM-vTPM and Keylime.

Chapter 3 investigates the feasibility of the SVSM-vTPM through an analysis of current public cloud offerings. Based on this evaluation, the project scope is refined to focus on providing the strongest attainable trust guarantees for cloud consumers.

Chapter 4 builds on the insights gained in Chapter 3 and documents the implementation of secure remote attestation in a public cloud setting. It follows an incremental development approach, discusses technical challenges, and explores automation and configuration improvements using supporting tools. It includes practical experimentation using Azure and infrastructure-as-code, aiming to provide maximal trust guarantees for cloud consumers.

For the final chapter, Chapter 5 concludes the thesis by summarizing the main findings, reflecting on the project's outcomes and faced challenges, and suggesting directions for future work.

Chapter 2

Background and Related Work

This section is structured to progressively introduce key concepts in confidential computing. It begins with an introduction to cloud computing and confidential computing, then gradually explores the technologies and mechanisms necessary to understand the design and implementation of SVSM-vTPM and Keylime, the centerpieces of this project. This section is a reflection of the initial literature study that has been conducted in the first 4 weeks of the semester. Any new concepts or literature encountered during the course of the project are discussed within the relevant chapters, where their context and relevance can be fully appreciated.

2.1 Cloud Computing

According to the National Institute of Standards and Technology (NIST), cloud computing is "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [14]. This model is the foundation of modern cloud services offered by major providers such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud. These cloud providers let individuals and companies rent their hardware infrastructure to deploy their workloads or store their data. They often work with a pay-as-you-go model, allowing users to avoid the risk of making big investments in servers and spending money on maintaining them. Other benefits include increased scalability and agility, as users can quickly respond to rising demand by deploying additional workloads on cloud nodes. Additionally, cloud services provide a high level of reliability and availability towards your customers, caused by the ability to deploy multiple instances of the same service and spread them more geographically. This reduces downtime, latency, and enhances disaster recovery capabilities, ensuring continuous service delivery for

businesses and customers [1].

To deliver these capabilities efficiently and securely, cloud providers rely heavily on virtualization technologies.

2.1.1 Virtualization

A **virtual machine (VM)** is a software-emulated version of a physical computer, that runs its own operating system and applications. It is managed by a hypervisor, software that virtualizes the server's underlying hardware resources and enables multiple VMs to share those resources while staying completely isolated from each other (see Figure 2.1). Cloud providers most commonly use VMs to deploy customers' workloads on their public infrastructure, as they provide high security through strict isolation. As a result, a workload on a VM is unable to interfere with the hypervisor or workloads of other VMs [15]. Although **container-based virtualization**, a virtualization method that shares the operating system and certain dependencies across containers, offers better performance, it lacks these strict isolation guarantees. Enhancing container isolation, through various techniques such as Security Namespaces [16] and Container-IMA [17], could improve its adoption.

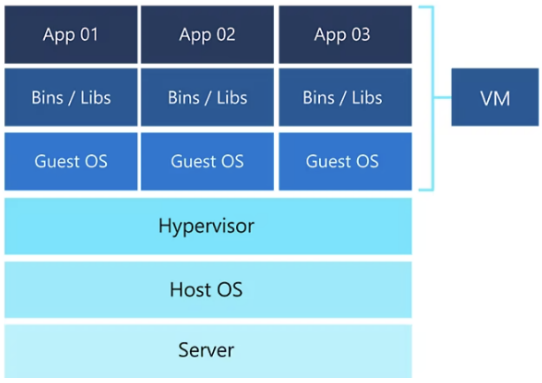


Figure 2.1: Hypervisor with multiple VMs
[15]

2.1.2 Cloud Security Issues

Although virtualization protects the cloud provider from the workloads and protects workloads from each other, it does not protect the consumer's workload from the cloud provider. Critically, data is executed unencrypted in memory, exposing it to potential inspection or tampering by the underlying hypervisor. This is especially concerning when deploying workloads that handle sensitive data, such as patient health records, intellectual property, or financial information.

Although these companies claim they offer great security, breaches and data leakage still occur. These issues are often caused by misconfiguration, exploits, and bugs introduced with new features. Consequently, companies or individuals that want to make use of the cloud, need to trust the cloud provider and its security measures for keeping their data confidential and executing their workloads with integrity.

Currently, the trust guarantees that consumers rely on stem from the cloud provider's reputation, Software Level Agreements (SLAs) or contractual obligations, and traditional software-based protections. However, these mechanisms are inherently limited. A provider's reputation offers no technical enforcement and cannot prevent insider threats or sophisticated attacks. SLAs and contracts may define consequences after a breach occurs, but they do not prevent the breach itself. Software protections, such as virtualization and access controls, can be bypassed if the underlying hypervisor is compromised, especially by privileged administrators. Moreover, software systems are inherently vulnerable as they have a large number of attack vectors and are continuously evolving. Collectively, these guarantees provide little assurance that workloads are truly protected from the cloud provider itself or other high-privilege threats.

2.2 Confidential Computing

Confidential computing tries to improve the trust guarantees discussed in Section 2.1.2 by introducing hardware solutions and attestation mechanisms. Together, these try to **remove the need for trust in the cloud provider** by ensuring full confidentiality and integrity of data in use.

These hardware solutions provide a more robust foundation to build trust on, known as a **hardware Root-of-Trust**. Unlike software, which is inherently malleable and susceptible to (remote) exploitation or tampering, hardware roots of trust have a rigid design and require physical access to compromise, making them significantly more resistant to subversion [9]. They are responsible for safeguarding the integrity and confidentiality of workloads by generating cryptographic proofs that can be externally verified, ensuring that data remains protected even during execution.

The cloud consumer can cryptographically verify these proofs generated by the hardware, in a process called **remote attestation**. If the proof is cryptographically incorrect or contains the wrong information, the consumer can decide not to deploy their privacy-sensitive data on that platform or abort current workloads.

Confidential computing enhances cloud security by providing encryption and integrity protection for applications at runtime, safeguarding them against memory

snooping and tampering, even from privileged insiders [8]. This is particularly critical as cloud infrastructure is increasingly used in privacy-sensitive domains, such as healthcare and finance, where stronger security guarantees are required [6]. Moreover, confidential computing supports compliance with evolving data privacy regulations, including the Health Insurance Portability and Accountability Act (HIPAA) [5] and the General Data Protection Regulation (GDPR) [4], which mandate strict protection of sensitive and personal data.

2.3 Trusted Execution Environments

One of the hardware solutions the field of confidential computing introduces is the Trusted Execution Environment. The TEE offers an isolated environment, often called an enclave, which is integrity-protected to prevent unauthorized access to computations and data. TEE-technology guarantees the authenticity, integrity and confidentiality of executed code, process states and memory [18]. They even offer resistance against attacks performed by a compromised hypervisor, an operating system and a physical attacker, effectively reducing the Trusted Computing Base (TCB).

The general architecture of a TEE divides the software layer, resident above the hardware layer, in two isolated environments depicted in Figure 2.2 . The untrusted area, also referred to as the **Rich Execution Environment (REE)**, hosts a normal OS and general applications, while the trusted area, also known as the Trusted Execution Environment, hosts applications that have the highest security requirements. Specialized hardware guarantees this separation [19].

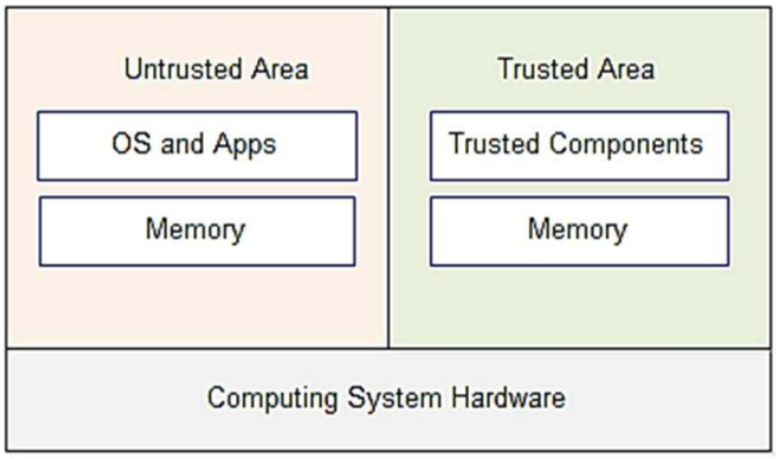


Figure 2.2: General architecture of TEE
[19]

A TEE typically gets realized by using a specialized CPU with an extended instruction set, which enables the code in a TEE to be executed in an isolated zone in the processor, isolating it from other processes. Although the definition of the TEE states that it provides confidentiality and integrity, in practice most real-world implementations predominantly focus on ensuring the confidentiality of code and data within the enclave. Some also enforce boot-time integrity to confirm the system starts in a known good state.

2.3.1 Intel Software Guard Extensions (SGX)

The first impactful hardware-assisted TEE was Intel Software Guard Extensions (SGX), offering a general-purpose security solution. This architecture introduces a new set of CPU instructions to create an enclave, that ensures confidentiality of the data at runtime by preventing memory snooping attacks [20]. An enclave is created by storing encrypted sensitive data and code of the application in the Enclave Page Cache (EPC), a small region in the main memory protected by strict access control mechanisms enforced by hardware [21]. Before the secured code is executed and the sensitive data is processed in an isolated zone of the processor, it gets decrypted by the **Memory Encryption Engine (MEE)** resident in the CPU (see Figure 2.3). Whenever this data leaves the CPU, the MEE will reëncrypt it. As a result, software such as the hypervisor or operating system cannot eavesdrop. Only the CPU and the SGX-related software components are part of the TCB [20]. Additionally, SGX provides mechanisms for local and remote attestation to verify code integrity, as well as a sealing mechanism to securely store and retrieve data in non-volatile memory like drives and disks [21].

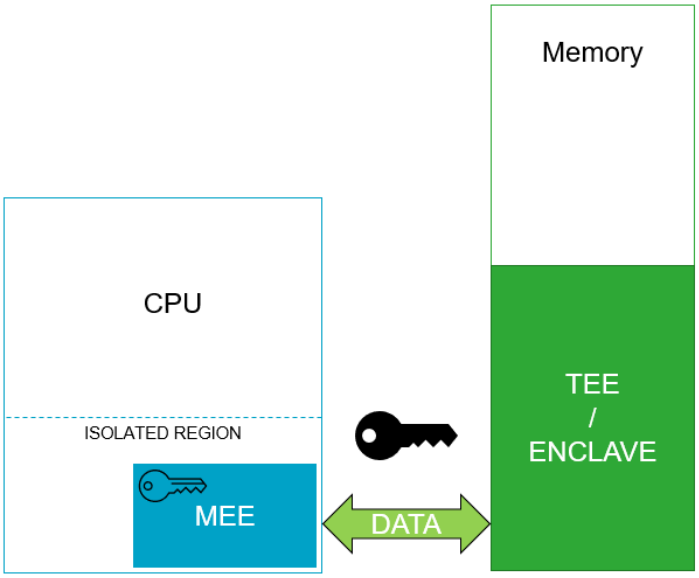


Figure 2.3: Diagram of SGX enclave

2.3.2 AMD Memory Encryption Technology

AMD Memory Encryption Technology introduces two groundbreaking hardware-based security features, AMD Secure Memory Encryption and AMD Secure Encrypted Virtualization. These features enable TEEs that are more applicable to public cloud infrastructure and infrastructure as a service (IaaS). It works similarly to SGX, where the MEE encrypts or decrypts the application's memory as it respectively leaves or enters the processor. It was created to prevent compromised hypervisors from analyzing and attacking guest VM memory space, as well as to defend against hardware attacks such as cold boot attacks. Key management is handled by the **AMD Secure Processor (AMD-SP)**, which runs a kernel signed by AMD. The main advantage of AMD TEEs over Intel SGX TEEs is that they require no application changes, making them transparent to the user [21].

Secure Memory encryption (SME) is a memory encryption feature designed to protect data stored in DRAM. It operates by utilizing an AES-128 encryption engine, integrated into the memory controller, that encrypts data when written to memory and decrypts it when read. The encryption key is generated at each system reboot and managed by the AMD-SP, ensuring that software running on the CPU cannot access it. Memory encryption is controlled via the C-bit in the page table entries, allowing selective encryption of memory pages like shown in Figure 2.4 . SME provides strong protection against physical attacks and requires no modifications to application software with limited performance overhead [22].

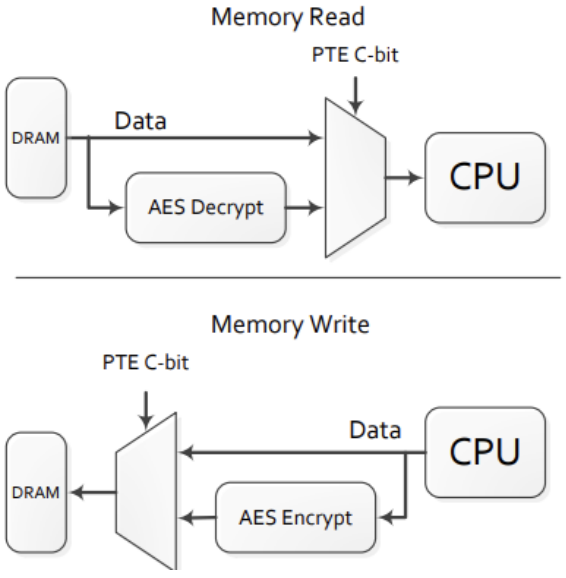


Figure 2.4: Optional memory Encryption with C-bit [22]

Secure Encryption Virtualization (SEV) is a security feature that integrates

SME with the AMD virtualization architecture, enhancing security in a cloud environment by supporting encrypted virtual machines [22]. Each VM is assigned a unique encryption key, managed by the AMD Secure Processor, ensuring cryptographic isolation between VMs and the hypervisor. Although the less privileged code (VM) relies on the high privileged code (Hypervisor) for deployment and execution, SEV ensures that the VM does not need to trust the hypervisor to maintain this isolation [8].

SEV-SNP (Secure Nested Paging) is an enhancement built upon SEV, to provide extra hardware-based security protections. Under SEV, a malicious hypervisor cannot read encrypted memory but can still modify it, potentially causing system crashes (DoS attacks) or injecting outdated memory states (replay attacks). SEV-SNP prevents these attacks by enforcing strong memory integrity protection to get a more isolated TEE. SEV-SNP protects memory integrity using two techniques: the Reverse Map Table (RMP) and Page Validation. The AMD-SP CPU uses the RMP to keep track of which VM owns each memory page, blocking unauthorized memory modification attempts. Page Validation adds an extra layer of security by making the guest system confirm memory assignments before using them [23].

Virtual Machine Privilege Levels (VMPLs) is an optional feature in the SEV-SNP architecture that allows users to partition the guest VM's address space into four isolated privilege levels. These layers are enforced through hardware isolation by adding the privilege level to the RMP and enable secure services or security controls to run at higher VMPLs, shielding them from both the guest OS and the host environment. VMPL0 holds the highest privilege, while VMPL3 is the least privileged [23]. For a modular VMPL-based design to be effective, a standardized calling convention is necessary to facilitate secure communication between the guest operating system and higher-privileged secure modules. **The Secure VM Service Module (SVSM)** realizes this by ensuring that interactions between different VMPL levels occur securely without interference from the untrusted host environment [24]. Residing at VMPL0, the SVSM first gets control of the VM, can initialize the VMPL layers, and can be used to implement secure services isolated from the rest of the VM [25]. Communication between VMPL1 and SVSM's VMPL0 context uses the Guest Hypervisor Communication Block (GHCB), a mechanism for exchanging messages via an encrypted shared memory page, which prevents a malicious hypervisor from intercepting that communication [24].

Another key feature of the SEV-SNP architecture is its support for hardware-based attestation through the **SEV-SNP attestation report**. This report proves the trustworthiness of the guest VM's **initial state**, such as the bootloader, firmware

and SVSM. To ensure the freshness of the report and prevent replay attacks, user-provided data can be injected into the report before generation.

2.4 Trusted Platform Modules and Remote Attestation

While TEEs protect the confidentiality of workloads by encrypting memory and isolating execution, most of them only ensure boot-time integrity but fail to ensure runtime integrity of the software running inside them. There is still a need to verify if the TEEs are not tampered with by an untrusted cloud provider or an attacker who gained administrative privileges [7]. The Trusted Platform Module (TPM) and remote attestation mechanisms provide ways to continuously prove that the system is running unmodified and in a known trusted state.

2.4.1 Trusted Platform Module (TPM)

The Trusted Platform Module is a tamper-resistant hardware chip that is described by [8] as a secure vault for storing cryptographic keys, certificates, and other secrets. It can perform asymmetric encryption, hashing operations, and even generate keys with its onboard Random Number Generator (RNG). Primarily, the TPM is used to safeguard the integrity of software and hardware running on the platform.

It does this by storing and authenticating cryptographic measurements (hashes) of applications/binaries inside of its **Platform Configuration Registers (PCRs)**. A PCR is a 20-byte register implemented in the volatile memory of the TPM, meaning its contents are cleared upon system reboot or power loss. PCRs can be collected by a remote third party and checked against an attestation policy containing the expected measurements, as shown in Figure 2.5. To prove their authenticity, the TPM generates a quote. This quote contains the selected PCR values and is cryptographically signed using a TPM-resident key.

This signing key is derived from the **Endorsement Key (EK)**, a unique asymmetric key embedded in the TPM's non-volatile memory (NV-memory) during manufacturing, accompanied by an **Endorsement Certificate (EC)**. The EC proves that the EK originates from a trusted hardware vendor. Consequently, if the TPM successfully decrypts a challenge encrypted with the EK's public part, it proves both its authenticity and the legitimacy of the generated quote.

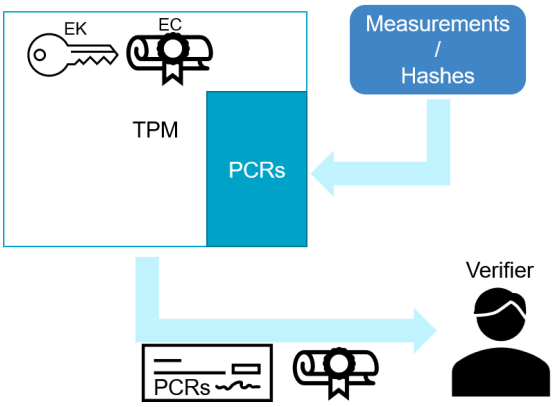


Figure 2.5: Simple diagram of attestation using the TPM

This way, the TPM serves as the hardware Root-of-Trust, where users only have to trust the CPU vendor to verify system integrity. This can be realized by building a chain of trust, where each part of the system can get its measurements authenticated by the TPM and verified against predefined policies, adding them to the set of trusted components.

Alongside the Endorsement Key (EK), the TPM also stores a Storage Root Key (SRK), which is used to encrypt (or wrap) other encryption keys that protect sensitive data. The SRK never leaves the TPM, ensuring it remains secure and uncompromised. When encrypted data needs to be accessed, the TPM first checks if the system is in a trusted state before unwrapping the encryption key with the SRK. This process, known as **sealing and unsealing**, ensures that protected data can only be decrypted when the system meets predefined security conditions [26].

2.4.2 Virtual TPM (vTPM)

In multi-tenant cloud environments, many customers may wish to verify the integrity of their virtual machines (VMs) simultaneously through remote attestation. However, TPM hardware does not scale well under such parallel demand, as operations like quote generation are relatively slow due to its reliance on low-bandwidth peripheral buses. Moreover, a physical TPM cannot provide isolated instances per VM. These limitations led to the development of virtual TPMs (vTPMs), first introduced by Berger et al. [27].

A vTPM is a software-based emulation of a physical TPM, designed to provide each virtual machine (VM) with its own isolated TPM instance that can perform quoting operations faster than a physical TPM while ensuring strict isolation. Unfortunately, the first designs either rely on trusting the host environment or rely on using the

hardware TPM for each operation [7].

Cloud-based vTPMs are vTPMs that are offered by cloud providers and that are managed by the hypervisor, enabling remote attestation on your confidential VMs. Although these should provide a way to verify the integrity of workloads running on the VM, they are often managed by the hypervisor and not chained to a hardware root of trust. Additionally, limited public knowledge about their implementation means they implicitly require trust in the hypervisor or cloud provider. Security flaws or design weaknesses in these vTPMs could compromise integrity. In contrast, **TEE-based vTPMs** place the vTPM within a Trusted Execution Environment, enhancing isolation and removing trust needed in the cloud provider.

2.4.3 Measured Boot

Measured boot is the process of recording the measurements of all binaries involved in the system's boot such as the bootloaders, firmware and operating system kernel. Cryptographic hashes of these binaries are recorded and extended[1] into the PCRs, essentially creating a log file that is protected and authenticated by the TPM. The log file can be requested by a verifier, forcing the TPM to generate a TPM quote. The verifier can then verify the signature to ensure it was generated by a trusted TPM and compare the values of the PCRs to the expected measurements (without malware). As a result, measured boot enables the detection of unauthorized modifications, allowing a verifier to confirm the system is in a trusted state before deploying services [7, 28].

Measured boot begins with the Core Root of Trust for Measurement (CRTM), a minimal, immutable firmware component embedded by the hardware manufacturer in non-writable memory that records a cryptographic hash of the next code into a TPM PCR, anchoring the chain of trust. Each subsequent component then measures the following component before executing it and extends that measurement into the PCR, thereby creating a continuous chain of trust through every stage of the boot process [29].

Often Measured Boot is confused with **Secure Boot**, since both involve boot-time protections, but they operate differently. In Secure Boot, each component verifies that the next one is correctly signed by a trusted authority and can halt the boot process immediately if a signature check fails. In contrast, Measured Boot computes

[1] An *extend* operation updates a Platform Configuration Register (PCR) as $\text{PCR}_{\text{new}} = H(\text{PCR}_{\text{old}} \,\|\, \text{measurement})$, where H is a cryptographic hash function and $\|$ denotes concatenation. This creates an immutable hash chain, ensuring previous measurements cannot be altered or removed.

and records cryptographic hashes of each component into TPM PCRs without enforcing any policy itself. Later, a separate verifier must compare those recorded measurements against an expected policy to detect unauthorized changes. Measured Boot offers greater flexibility and auditability by logging each boot stage's measurements for later or remote verification, rather than simply enforcing immediate signature checks [29].

2.4.4 Integrity Measurement Architecture (IMA)

Although Measured Boot ensures that a system starts in a trusted state (boot-integrity), it does not ensure it remains in this trusted state (runtime-integrity). For example, an untrusted cloud provider or attacker who gains administrative privileges could dynamically load malicious kernel extensions or downgrade security critical subsystems to perform runtime exploitation.

To address these runtime integrity concerns, Sailer et al. proposed Integrity Measurement Architecture (IMA) [30]. The architecture of IMA is displayed in Figure 2.6. The IMA hooks detect when measurement events occur, such as executing a binary, loading a kernel module, or a root user opening a file. The Measurement Agent then captures this event's file path and computes the cryptographic hash of the content of that file. Subsequently, it stores these values in the Measurements List (ML) and extends them into the TPM's PCRs. The Measurements List provides a detailed history of measurement events, which is critical because the PCRs only contains a cumulative hash due to its limited size. This list can be used by the verifier to reconstruct the PCRs and verify if the TPM contains the correct values. This design allows the verifier to focus on a subset of measurements of interest without requiring expected values for every single event. Similar to the measured boot, a remote verifier can request a signed TPM quote together with the ML, and check the integrity of the system [17].

There remains one question: why can we trust IMA to perform correct measurements? This is because users can trust its design and implementation as it has been extensively researched, and its integrity is protected by the measured boot process (see Section 2.4.3).

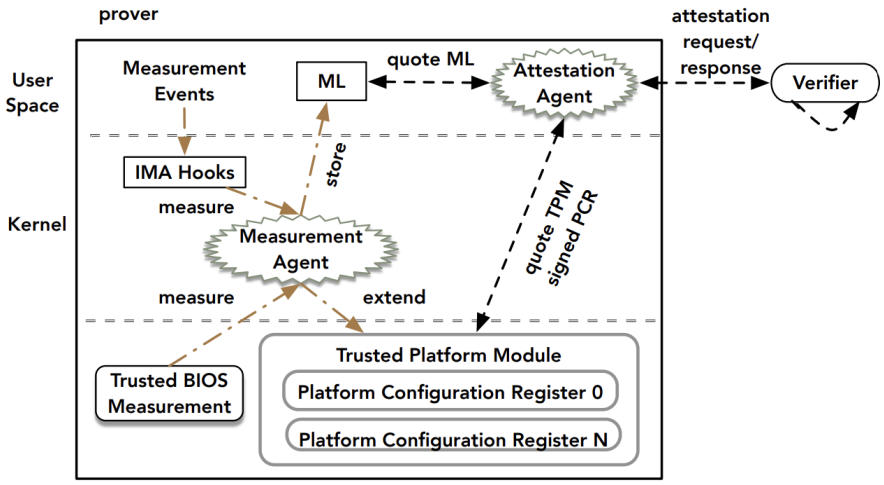


Figure 2.6: Integrity Measurement Architecture [17]

2.4.5 Keylime: A remote attestation framework

Keylime, introduced by Schear et al. [31], is an open-source, scalable trusted cloud key management system that serves as a remote attestation framework. It provides a way to bootstrap hardware-rooted cryptographic identities for a fleet of Infrastructure-as-a-Service (IaaS) nodes and enables their remote attestation. It interfaces with the TPM or vTMP available on the nodes or VMs, using them as a root of trust. Because Keylime is an open-source project available on Github [32], organizations and developers can customize attestation workflows to their unique environments. Additionally, Keylime was accepted into the Cloud Native Computing Foundation (CNCF) Sandbox in September 2020, and companies like IBM have integrated Keylime into their cloud infrastructure to enhance security measures [33]. To perform both boot-time and runtime attestation efficiently on a fleet of systems, Keylime's design is divided into four major components: the verifier, registrar, agent, and tenant. These components securely communicate over mutual TLS (mTLS) and require different cryptographic and TPM-related dependencies to run [11].

The Keylime agent is installed on every node that needs attestation. First, the agent must register with the registrar to establish trust, as shown in Figure 5, which illustrates the registration protocol. This agent first sends the node's UUID, endorsement key (EK)[2], endorsement key certificate (EC), and the proposed attestation identity key (AIK)[2] [7]. With the EC and a challenge encrypted by the

[2]If the EK or AIK are mentioned outside of internal TPM signing operations, it typically refers to EK_{pub} or AIK_{pub}, as extracting the private keys from the TPM should not be possible.

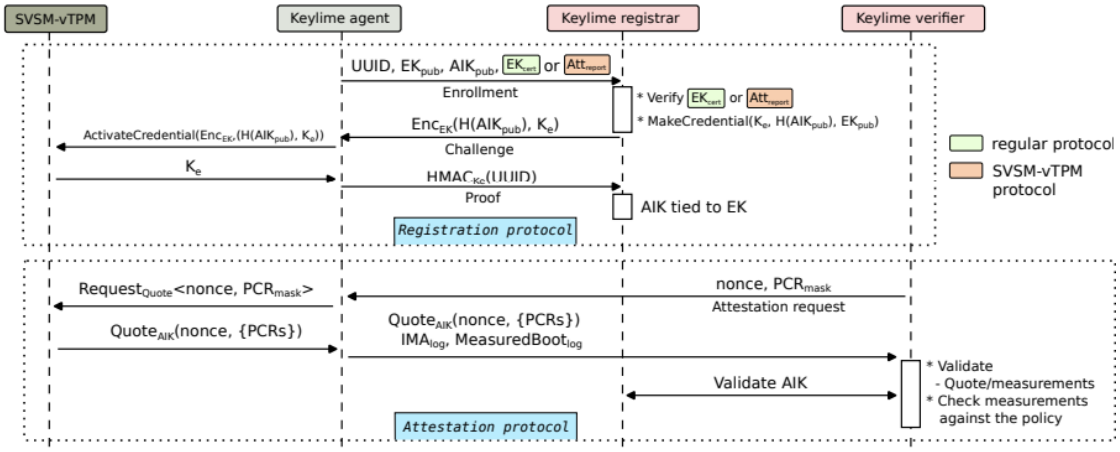


Figure 2.7: Keylime's Registration and Attestation protocol [7]

public part of the EK, the registrar can verify that the agent is running on trusted hardware, as explained in Section 2.4.1. The AIK is used to sign TPM-generated quotes, ensuring the integrity and authenticity of the reported system state without exposing the EK, which remains solely for identity verification.

The registrar, hosted in the cloud or on private infrastructure, holds a registry of trusted nodes and can be queried by the verifier to check if a node is trusted. It contains preinstalled certificates of trusted CPU-vendors, which it uses to check if the node's agent is running on a node with a trusted TPM-chip before appending it to the registry of trusted nodes. The registrar also provides an interface for node revocation. This is needed when an agent fails to provide a correct attestation report when requested by a verifier, so applications using that node can get notified and stop interacting with it [11, 31].

The verifier performs the attestation of an agent and issues revocation messages if the agent exits the trusted state. The attestation process, as shown in Figure 2.7, happens periodically to verify measurements from the boot log or the IMA log. The attestation requests contain a PCR mask, which is a list of PCR values the verifier wants to be included in the quote. Additionally, a nonce is added to the request to prevent replay attacks by including it in the signed-quote. The verifier then validates this nonce, checks if the right AIK is used for the signature, and the measurements match the expected measurements defined in a policy. This way, the verifier can keep the integrity of the fleet of nodes checked [7, 11, 31].

The fourth component of Keylime, the tenant, initiates the attestation by requesting the verifier to assess the integrity of nodes running the agent. The tenant also plays a crucial role in securely provisioning nodes by employing secret sharing techniques

discussed in [31].

2.5 The SVSM-vTPM

Narayanan et al. [7] proposed and implemented a new vTPM design that virtualizes the hardware root of trust while also ensuring that no trust is required in the cloud provider. To avoid relying on the cloud provider, they leverage the SEV-SNP hardware (see Section 2.3.2) to execute the vTPM in a TEE. Additionally, the design takes a new approach on vTPM state management, proposing a stateless ephemeral design. As a result, it allows each confidential container to have its own private instance of a vTPM isolated from the hypervisors and other VMs. This ephemeral design completely eliminates the classes of attacks that come with state protection. The ephemeral vTPM, also called the SVSM-vTPM, is designed around 3 security requirements: isolation, secure communication, and protected state [7].

2.5.1 Threat model

This design offers protection to an attacker with the highest privilege over the software or firmware and even to physical attackers, given that we trust the AMD secure processor hardware and the SEV-SNP and SVSM implementations. Additionally it protects from attackers compromising applications or services running on the guest VM by exploiting vulnerabilities on those services through network interaction. This is realized by leveraging Virtual Machine Privilege Levels (VMPLs) to isolate the vTPM from the Guest OS in the same confidential VM. IMA-attacks, cyphertext side-channel attacks, and runtime attacks exploiting overflow mechanisms are out of scope [7].

2.5.2 Isolation

The design must isolate the vTPM from the entire host environment, to protect it from reading sensitive data or compromising execution integrity. For this, the design uses TEEs to host the vTPM, as cloud vTPM and other alternatives do not fully guarantee this. Not only will the AMD SEV-SNP technology provide this TEE environment, but it will also enable the isolation of the vTPM from the Guest OS by using the optional VMPLs (explained in Section 2.3.2).

In most other vTPM designs, the architecture relies on a vTPM-manager, which is responsible for creating the vTPM and rooting it to real trusted hardware like

the TPM. Additionally, this manager handles the communication between the guest VMs and the vTPM and handles the vTPM state during its lifecycle. As a result, the manager becomes a single central point of failure and is often targeted by attackers. Attacks range from simple denial-of-service attacks to secret exfiltration attacks and collusion attacks. The SVSM-vTPM mitigates these attacks by providing each confidential VM with its own vTPM, ensuring the vTPMs stay completely isolated [7].

2.5.3 Secure communication

Targeting the communication between a Virtual Machine (VM) and its virtual Trusted Platform Module (vTPM) creates a potential attack vector. An attacker could compromise measured boot and runtime attestation by altering the request and response buffers. Developing specialized protocols, such as TPM HMAC or the DMTF Security Protocol and Data Model (SPDM), is complex, leading to limited adoption in vTPM implementations. Previous TEE-based vTPM designs often rely on Transport Layer Security (TLS) for communication, which requires modifications to the software stack and consequently increases the Trusted Computing Base (TCB) [7].

The SVSM-vTPM is hosted in the guest VM, so it can rely on the **SVSM** mechanism to securely pass execution between VMPLs, eliminating the need for implementing a secure communication protocol. The guest OS communicates with the vTPM using the **GHCB** explained in section 2.3.2. The guest OS can use this block/buffer to communicate the TPM request in clear text, which triggers an exit into the hypervisor. For the hypervisor, this page remains confidential and integrity-protected because of SEV-SNP, which means the messages remain protected. Additionally, AMD SEV ensures that the hypervisor can only resume execution of the VM at the correct privilege level, making it unable to suppress request and response messages without halting the VM. When it resumes execution at VMPL0, the vTPM can read the command and respond in a page in memory, giving control back to the lower privileged VMPLs in a similar way [24, 7].

2.5.4 State

A physical TPM needs to store some values in non-volatile storage such as the EK, the SRK, the EC, NV Index values, user-made objects, and saved state when TPM is shutdown. As a result, these values remain accessible to the TPM after system restart. A vTPM does not have a dedicated non-volatile storage as it resides in software. Therefore, a lot of vTPM designs typically implement non-volatile storage

in a file on disk. However, this approach presents several challenges, such as securely storing the secret key used to encrypt that file and injecting that secret with the encrypted TPM state into the vTPM during boot-up [7].

Instead, the SVSM-vTPM, does not store state at all and generates a fresh set of seeds and keys on every boot. This approach makes the vTPM easier to implement and safer because it avoids all the problems related to persisting state, like state injection and safeguarding the encrypted state file. However, this technique poses some downsides, like the inability to store secrets across reboots when using the sealing mechanism, explained in Section 2.4.1 [7].

2.5.5 Implementation

The implementation of the ephemeral vTPM follows AMD's recommended software stack, including QEMU, OVMF, and the Linux kernel, all modified for SEV-SNP support. Minor changes to Keylime were made for remote attestation and the SVSM implementation got extended with a minimal C library, WolfSSL for cryptographic functions, and Microsoft's TPM reference implementation for TPM 2.0. These changes were minimal to keep the TCB as small as possible [7]. The Proof-of-Concept, available on Github [12], was tested on Cloudlab. Cloudlab is an experimental bare-metal cloud environment, offering full customizability of cloud infrastructure for research purposes [34].

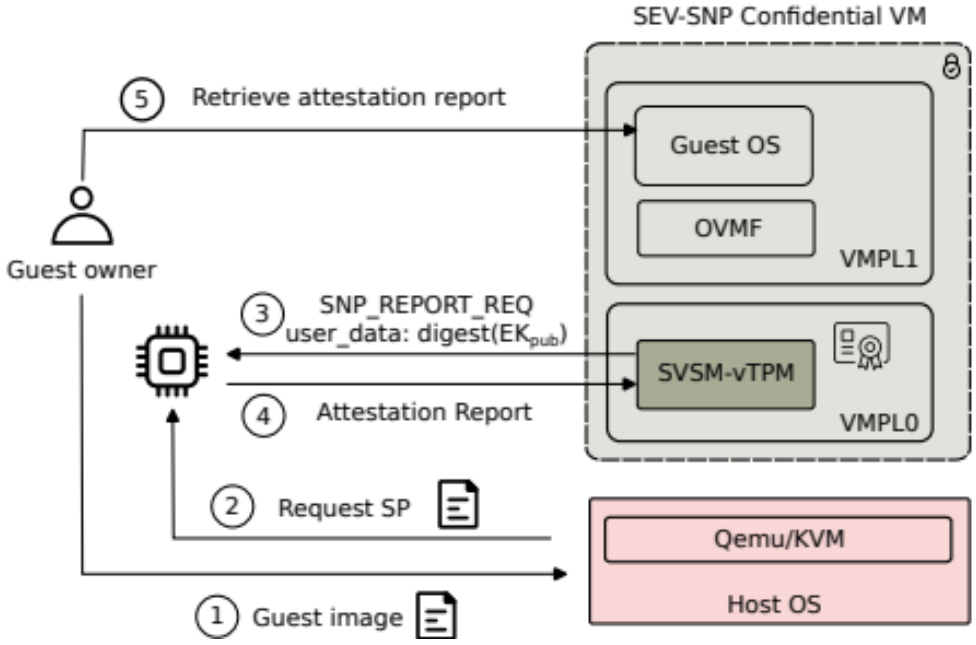


Figure 2.8: SVSM-vTPM implementation
[7]

To **add the SVSM-vTPM to the hardware chain of trust**, the SVSM-vTPM must interface with the AMD-SP, like shown in Figure 2.8. First, the guest image is loaded into a VM and its contents are measured by the AMD-SP (1 & 2). Once the guest image is loaded, the hypervisor passes control to the SVSM firmware, which initializes the guest CPU, memory, page table, and the vTPM. The vTPM then requests a SEV-SNP attestation report (Att_{rep}) from the AMD-SP (3), which provides a signed quote over the boot measurements of the VM, the VMPL level, and some user-provided data (4). The VMPL level can be verified by the AMD-SP because the request made by the vTPM is encrypted with a key uniquely possessed by that VMPL level and the AMD-SP, which is the VM platform communication key (VMPCK). The field for user-provided data gets populated with a digest of the ephemeral EK. This way the Att_{rep} can serve as an EC to verify the EK, because the vTPM does not possess a EC. This is because a vTPM is not a real hardware device and there is no manufacturer to create a certificate for this EK. To perform remote attestation (5), a keylime agent is ran on every VM that needs attestation. To make Keylime work with the SVSM-vTPM, changes need to be made to the registration protocol, shown in Figure 2.7. Instead of the agent providing the EC, it now provides the Att_{rep} generated by trusted hardware because it has no EC. Consequently, the registrar needs to make some changes on how to verify the registering guest VM. First, it needs to verify that this report is signed using AMD's versioned chip endorsement keys (VCEK), to make sure it is running on genuine AMD hardware. Subsequently, it should verify if the digest of the EK_{pub} equals the digest signed in the Att_{rep}. Then it needs to check the measurements against the expected measurements to make sure the SVSM-vTPM is not tampered with. Lastly, it needs to verify if the reported VMPL level of the signed Att_{rep} equals 0, making sure that it is not communicating with a fake vTPM running in a lower privilege level. Other Keylime protocols and the rest of the register protocol do not need any changes to use the ephemeral vTPM [7].

Performance tests showed that this implementation is up to 5× faster than a QEMU-hosted vTPM. This is because the SVSM-vTPM runs inside the guest VM, eliminating the need for communication with separate vTPM hosted elsewhere. A security analysis discusses how this implementation prevents certain attacks and vulnerabilities such as booting with a fake vTPM, providing a fake SEV-SNP attestation report, starting the confidential VM with no SVSM module, and weaknesses in the random number generator (RNG). A more in-depth discussion of the performance and security analysis is available on [7].

Chapter 3

Feasibility of The SVSM-vTPM

This chapter investigates the feasibility of deploying the SVSM-vTPM design on current public cloud platforms, a critical question to assess its industry-relevance. To answer this, a structured methodology is applied to define deployment requirements, analyze existing platforms, and evaluate their capabilities based on publicly available information. The results demonstrate that none of the major cloud providers currently support the necessary features, leading to a pivotal change in the research direction. Based on these findings, a comparative analysis of existing cloud-based vTPM offerings is conducted to identify the most suitable alternative platform for implementing secure remote attestation.

3.1 Methodology

To document and assess the feasibility of the SVSM-vTPM design in public cloud, it is important to clearly define a step-by-step approach that will be used to guide this research.

First, building on the research of the SVSM-vTPM already presented in the background, we will conduct a detailed analysis of the Proof-of-Concept implementation published on GitHub [10]. This analysis will focus on identifying and understanding the specific modifications made to the original components and the artifacts involved in realizing the SVSM-vTPM design.

Next, based on insights from the initial analysis, we will define a set of high-level deployment requirements. These requirements will outline what a public cloud provider must support to enable SVSM-vTPM deployment. The rationale behind each requirement will be justified.

Subsequently, these requirements will be validated against the most popular cloud platforms. This step involves thoroughly studying each platform's offerings and

capabilities to assess whether they satisfy the defined criteria.

Lastly, the design will be incrementally implemented on the platform that meets all of the defined criteria. This implementation process will extensively document all challenges encountered during implementation. If the feasibility is not fully established in the earlier phases, this final step will serve as the conclusive determinant.

3.2 Feasibility Assessment Results

3.2.1 PoC Analysis

To implement the SVSM-vTPM as a Proof-of-Concept (PoC), artifacts like the SVSM, OVMF, QEMU, Linux, and Keylime were used and modified. The reason for using these artifacts and their modifications was discovered by thoroughly going over every commit in the repository and will be explained [12]. It is important to note that [7] claims that the changes discussed for QEMU, OVMF, and the Linux Kernel for both host and guests will eventually be upstreamed.

Linux-SVSM

The Secure VM Service Module, discussed in Section 2.3.2, is the core component of the SVSM-vTPM PoC, acting as a minimal module that is modified to host and instantiate the virtual TPM within VMPL0.

To mediate TPM requests and responses with the VMPL1 guest, they modify the SVSM to implement a CRB(Command/Response Buffer)-based protocol, similar to other vTPM implementations. A CRB protocol uses two memory-mapped buffers (one for commands, one for responses) and a few control registers to coordinate each request/response handshake. The CRB registers get defined and created in the `/src/vtpm/init.rs` file. To handle these CRB-requests, a function `handle_vtpm_request` in `/src/svsm_request.rs` is added to implement a CRB state machine.

A minimal C runtime library was integrated alongside the WolfSSL library in `build.rs` to support cryptographic operations such as hashing. To adhere to established standards without developing a TPM from scratch, Microsoft's TPM 2.0 software reference implementation was also added as a dependency. Within this setup, the `send_tpm_command` function located in `/src/vtpm/init.rs` facilitates the transmission of raw TPM commands from the command buffer to this TPM library, capturing and returning the results.

To manufacture the vTPM, the EK was created and written to NVRAM, in the `/src/vtpm/manufacture.rs` file. It does not write an EK certificate to NVRAM, as the EK has just been created and can not be signed by a real hardware manufacturer. Additionally, the new manufacturer file also adds functionality to return the public EK.

Finally, functionality was also in `/src/vtpm/report.rs` added to request a SEV-SNP attestation report and store it in the non-volatile (NV) index where the EK certificate normally resides. This request will try to populate the user-data with a hash of the public EK, so the vTPM can be chained to a hardware root of trust, as explained in Section 2.5.5 [7, 35].

OVMF

Open Virtual Machine Firmware (OVMF) is an open-source Unified Extensible Firmware Interface (EUFI) implementation designed to run inside virtual machines under QEMU. Acting as the VM's "BIOS," it initializes virtual hardware, provides EFI services (console, block devices, networking), and hands control off to the guest OS's bootloader.

In the SVSM-vTPM PoC, OVMF is the firmware layer loaded into VMPL1. When the guest OS or bootloader issues a TPM-related call, OVMF routes that request through the CRB interface exposed by the SVSM in VMPL0.

This artifact was modified to support the SEV-SNP architecture. Furthermore, changes were necessary to implement a "doorbell" mechanism, ensuring the SVSM (operating at VMPL0) was notified when the guest (at VMPL1) placed messages in the shared buffer protected by SEV-SNP. This was necessary because the vTPM and the guest are not actively polling the shared buffer to find new messages [35]. This was realized by extending the CRB implementation to send TPM requests via the MSR protocol and extending the VMGEXIT with an extra parameter in `OvmfPkg/Library/VmgExitLib/VmgExitSvsm.c` [7, 12, 35].

QEMU

Quick Emulator (QEMU) is an open-source machine emulator and virtualizer that provides a complete virtual hardware environment for VM's. When used in conjunction with the Linux KVM kernel module, QEMU can run guest code directly on the host CPU for near-native performance, while still emulating peripherals such as disks, network interfaces, and firmware.

In the SVSM-vTPM PoC, QEMU's job is simply to launch the confidential VM

under SEV-SNP (loading OVMF and the SVSM binary into the measured, encrypted guest), and then hand execution off to the SVSM module running in VMPL0, which takes over all TPM emulation and command handling. A modified QEMU was used to support the AMD SEV-SNP architecture [7]. Furthermore, it gets modified to add a new CRB interface that contains the boilerplate code such that the entirety of CRB is handled in the SVSM, and does not pass through the hypervisor and Host OS [12].

Linux Host and Guest

The Linux host runs under QEMU/KVM, using the standard SEV-SNP configuration without any additional OS modifications [7, 12].

The Linux guest runs in VMPL1 and uses its standard TPM driver to talk to the vTPM over the CRB buffers. A small patch in `/drivers/char/tpm/tpm_crb.c` makes each CRB access go through the SEV GHCB, explained in Section 2.3.2, which ensures secure communication protected from the host environment [12, 7].

Keylime

As noted in Section 2.5.5, Keylime's framework was altered to support the remote attestation of the SVSM-vTPM. This section will shortly discuss where these changes can be found and why they are made.

The registrar's `do_POST` function, in charge of registering an agent, was extended to accept and validate the AMD SEV-SNP attestation report as an alternative to an EC. This is done by validating the schema and verifying that the hash embedded in the report's user-data section exactly matches the public EK submitted by the agent on registration [12].

Keylime's tenant component, which orchestrates agent enrollment with the verifier and checks that the EK certificate chains to a trusted root, was also enhanced to ensure the attestation report was generated in VMPL0 and is signed by the appropriate VCEK certificate. These checks were added to the `check_ek` function in `/keylime/tenant.py`, and use a new helper function `verify_ekcert_surrogate`. Unlike regular EK certificates, where the issuer is trusted via Keylime's local certificate store, the VCEK must be retrieved from AMD's key distribution service, since it is unique to each processor and firmware version and may be updated or revoked over time if vulnerabilities are discovered [12, 23].

3.2.2 Requirements for implementation

If someone were to implement the SVSM-vTPM on a public cloud, the platform would have to adhere to some requirements. This set of high-level deployment requirements is derived from its design and modifications, discussed in Section 2.5 and 3.2.1.

1. **Hardware Requirements**

   The platform must offer VMs that use AMD SEV-SNP hardware, as the design of the SVSM-vTPM is based entirely on leveraging this hardware to ensure its security properties and root itself in trust. As discussed in Section 2.5, this hardware ensures the SVSM-vTPM properties like host isolation, guest isolation, and secure communication through creating a TEE with hardware-isolated VMPLs. Additionally, the AMD SEV-SNP attestation report is used to chain the SVSM-vTPM to a hardware Root-of-Trust.

2. **Custom Service Hosting in VMPL0**

   As a cloud consumer, you need the capability to customize or deploy services within VMPL0, because the implementation of the SVSM-vTPM, as detailed in the Section 3.2.1, required direct changes to the SVSM itself. These include adding TPM support, cryptographic libraries, and an attestation interface, all of which must operate securely within VMPL0. These abilities guarantee the vTPM's isolation from the guest and provide the ability for secure communication discussed in Section 2.5.

3. **Firmware & Hypervisor Flexibility**

   The cloud platform must allow consumers to patch or supply their own OVMF firmware and make modifications to the hypervisor. This requirement stems from the SVSM-vTPM implementation described in the modifications section, where OVMF was adapted to implement the VMPL1-to-VMPL0 doorbell signaling mechanism, and QEMU was modified to establish the shared encrypted memory region used for CRB communication. These components are essential for enabling secure, low-level communication between the guest and the SVSM. Without the ability to modify the firmware and hypervisor, integrating such mechanisms into the trusted computing base would not be feasible. Alternatively, if the cloud provider already provisions such a buffer and signaling mechanism, this flexibility could be relaxed.

4. **Guest-Level Orchestration**

   Tenants must be free to install their own guest OS as some changes might be required, like discussed in 3.2.1. Additionally, a cloud consumer must

be able to install modified Keylime's components within the guest VM to support custom attestation workflows. Furthermore, the platform must permit secure communication (e.g. mTLS) between the agent, tenant, and verifier, as Keylime needs this to implement secure remote attestation.

3.2.3 Evaluation of Public Cloud Providers

In this section, the requirements defined in Section 3.2.2 are used as the basis for evaluating Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform. For clarity and traceability, these requirements are referenced throughout using numbered labels such as (1), (2), etc. The selected platforms represent the most widely adopted and mature public cloud providers currently available [36]. This process aims to prove whether the design of the SVSM-vTPM is currently feasible on these platforms.

Amazon Web Services (AWS)

AWS supports AMD SEV-SNP on specific EC2 instance types (M6a, C6a, R6a) in select regions (e.g., US East, Europe) [37]. This enables the creation of confidential VMs with hardware-based isolation (1).

AWS does not seem to expose the capability for consumers to execute code at VMPL0 (2). This limitation is evident in the available documentation, which outlines the use of SEV-SNP but does not mention support for VMPL0 execution or the deployment of SVSM by consumers [37]. Additionally, the API only provides functionality to run an SEV-SNP enabled instance, but does not provide functionality to use VMPLs [38].

AWS EC2 instances utilize OVMF for booting, and while the source code for the UEFI firmware is publicly available on GitHub [39] , AWS does not mention the capability to modify the OVMF (3). Additionally, their very efficient lightweight hypervisor, called the Nitro hypervisor, is a closed-sourced software that does not allow any changes or hook implementations from the cloud consumer (3)[40].

On AWS, you are free to install and modify any custom software (e.g. Guest OS and Keylime) inside your EC2 guest VM and communicate over HTTP or mTLS just as on a bare-metal host (4) [41].

Microsoft Azure

Azure's Confidential VM families DCasv5, DCadsv5, ECasv5 and ECadsv5 are built on 3rd Gen AMD EPYC processors and offer SEV-SNP hardware-backed encryption and integrity protection for in-use data [42]. This way it makes the required hardware available to the consumer (1).

Azure's Confidential VMs do not let customers insert arbitrary binaries into the VMPL0 (2). Azure reserves the VMPL0 for its own **Host Compatibility Layer (HCL)**, also referred to as guest paravisor. This is a closed-source virtualized firmware environment running within the context of the confidential VM, that hosts para-virtualized devices including their own vTPM [43, 44, 45].

Azure does not permit tenants to modify the OVMF or the underlying hypervisor in its standard virtual machine offerings, including Confidential VMs (3). While Azure allows for certain customizations within the UEFI environment, such as modifying Secure Boot keys in Trusted Launch and Confidential VMs, this capability does not seem to extend to modifying OVMF firmware itself [46]. Additionally, Azure's hypervisor is managed exclusively by Microsoft, preventing consumers from making modifications [47].

Azure allows consumers to boot a confidential VM with a custom guest OS by using images [48]. Afterwards, software like Keylime can be installed, and the needed communications can be allowed (4).

Google Cloud Platform (GCP)

GCP supports AMD SEV-SNP with their N2D machine types with the AMD EPYC Milan CPU in Europe, Asia and the US (1) [49].

No official source from Google states that consumers can use VMPLs on GCP to host secure services in a higher privilege level, this suggests that the optional VMPL feature of the hardware is not used (2) [49, 50, 51].

Because Google locks and signs the OVMF firmware to help establish that the CVM is running on genuine Google-managed firmware [52], customers cannot inject their own SVSM or make changes to the OVMF (3).

Google Cloud's CVMs allow consumers to host their own guest operating system by creating custom images with the necessary configurations [53]. Once their custom OS is deployed, they have full control to install software like Keylime and configure in- and outbound communications (4).

3.3 Research Pivot Based on Findings

The SVSM-vTPM design and the modifications to its artifacts were studied to define and justify a set of high-level deployment requirements. From the result of the third step of the methodology, discussed in Section 3.2.3, it can be concluded that none of the specified cloud providers meet the criteria to deploy the SVSM-vTPM. This theoretically proves that **the design is not currently feasible in public cloud offerings**, and that there is no possibility to execute the fourth step of the

methodology (implementation). This outcome directly addresses the first project goal outlined in Section 1.3.

Although the studied cloud providers all offer support for CVMs using AMD SEV-SNP hardware and allow custom guest code inside that CVM, they lack the ability for consumers to host custom secure services in VMPL0. This requirement was very critical to the feasibility of the SVSM-vTPM, as most changes were made to the SVSM to implement the vTPM in VMPL0, to ensure the desired security properties like host isolation, guest isolation, and secure communication. A likely reason for this limitation is rooted in cloud provider design principles: exposing VMPL0 would effectively grant consumers low-level hardware control, undermining hypervisor management, breaking standardized security boundaries, and significantly complicating attestation and support.

It should be noted that the evaluation, performed in Section 3.2.3, relied solely on publicly available documentation from multiple sources to evaluate cloud provider capabilities. Due to access restrictions and paywalls, no direct testing on actual platforms was possible. Therefore, some platform features or future changes may not be fully reflected.

During this study, it was discovered that Azure's vTPM design shares significant architectural similarities with the SVSM-vTPM, particularly its implementation in VMPL0. This unexpected finding prompted **a strategic pivot in the research direction**: instead of pursuing the original SVSM-vTPM implementation, which current public cloud constraints render infeasible, the focus shifted toward a comparative analysis of existing cloud-provided vTPMs. The motivation behind this pivot was to evaluate how closely these existing solutions align with the intended security properties of the SVSM-vTPM design. By identifying the most suitable alternative, the study seeks to provide a viable path forward for implementing secure remote attestation within the limitations of current cloud infrastructure.

3.4 Comparative Analysis of Cloud vTPMs

As concluded, the SVSM-vTPM is not currently viable in public cloud offerings. However, this does not mean it is not industry-relevant. This work will use it as a golden standard to find the most similar and secure cloud-offered vTPM. It is important to note that we look at vTPMs offered with the CVMs using AMD SEV-SNP.

3.4.1 Selected Properties for Analysis

The security properties of the SVSM-vTPM, discussed in Section 2.5 , are very desirable as they effectively remove trust in the cloud provider. They will be used to determine the most secure vTPM available in public cloud. The desired properties:

- **Isolation from Host:** The vTPM must be fully isolated from the host environment (hypervisor and host OS) through putting it inside an integrity-protected TEE. Software protection (e.g. access control) is not deemed enough, as software is a weak root of trust.
- **Isolation from Guest:** Whether the vTPM is shielded from tampering or misuse by the guest virtual machine.
- **Backed by Hardware Root-of-Trust:** The vTPM is anchored in a hardware root-of-trust.
- **Secure Communication:** Whether the vTPM is hosted on the same CVM as the guest, so the communication between them is shielded from third parties. Communication with an external vTPM using an extra communication protocol, is seen as partially secure, as this introduces a bigger TCB.
- **Open source:** Whether the design and implementation are fully known. This way, it can be audited for insecure implementation or backdoors.
- **Guest attestation support:** The vTPM provides a TPM interface that allows cloud consumers to perform attestation independently. Consumers can verify the integrity of their VMs without relying on cloud provider-managed attestation services
- **Stateless:** The vTPM keys are generated on each VM boot, which provides resistance to attacks on persistent state. However, this property is undesirable when consumers require key persistence for sealing data across reboots (often the case). As a result, we consider it the least important property.

3.4.2 Platform Comparison

Each provider's vTPM Implementation was evaluated against the properties, defined in Section 3.4.1, to identify which platform offers the strongest trust guarantees (see Table 3.1).

Both the nitroTPM [54] from AWS and the vTPM provided by GCP [55] , are implemented on the cloud provider's hypervisor outside of the CVM, making them susceptible to tampering. Although this does ensure isolation from the guest code

Table 3.1: Comparison of cloud vTPMs against SVSM-vTPM's security properties

vTPM Property	SVSM-vTPM	AWS	GCP	Azure
Isolated from Host	✓	✗	✗	✓
Backed by Hardware	✓	✗	✗	✓
Isolated from Guest	✓	✓	✓	✓
Secure Communication	✓	♦	♦	✓
Open Source	✓	✗	✗	✗
Guest Attestation Support	✓	♦	✓	✓
Stateless	✓	✗	✗	✗

✓ Present ♦ Partial ✗ Absent

through virtualization, it means AWS and GCP have to set up secure communication protocols, which could be subject to various attacks and ultimately increase the TCB.

Furthermore, neither AWS nor GCP vTPMs are backed by a hardware root-of-trust. GCP vTPM keys are backed by a Google-issued certificate [56, 57], while Amazon does not seem to provide any cryptographic certificates for NitroTPM [58, 59]. Both providers do offer the ability to generate AMD SEV-SNP reports, which provide hardware-backed attestation of the platform's boot-time integrity [55, 60]. However, this mechanism is limited to verifying the system's initial state at launch. It does not replace the broader security functionality of a vTPM, which together with IMA enables runtime attestation.

Azure's vTPM design (see Figure 3.1), on the other hand, hosts its vTPM inside of an AMD SEV-SNP TEE, providing full isolation from the host. Much like the SVSM-vTPM, Azure implements the vTPM in the VM together with the guest application, making communication between the guest and vTPM secure through the TEE protections. To remain isolated from the guest, it also implements its vTPM design in a higher privilege level [43]. Furthermore, Azure's vTPM is backed by the AMD SEV-SNP hardware in a similar way to the SVSM-vTPM, leveraging the Attestation report to verify it's integrity and authenticate its keys [61]. As of right now, not a single provider has revealed the source code of their vTPM, requiring you to trust that it has no backdoors or other insecurities. Azure is planning on open-sourcing their Host Compatibility Layer (HCL), where the vTPM is implemented, which could greatly increase its credibility [62].

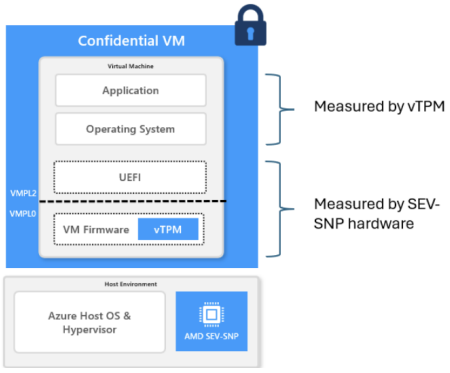


Figure 3.1: Azure's vTPM design
[43]

All three major clouds offer attestation services (Google Cloud attestation, Azure Attestation, Nitro), but you still have to trust each provider's verification software. True autonomous guest attestation needs a full TPM interface. Azure and GCP expose guest-attestation APIs so you can fetch and validate vTPM measurements yourself [45, 55]. AWS's Nitro vTPM is more limited and doesn't let you retrieve the vTPM's certificate for independent validation [59].

Lastly, all providers document that you can use the vTPM for sealing secrets [54, 61, 63], which implies that they are not stateless and susceptible to attacks on state.

3.4.3 Analysis Conclusion

AWS and GCP vTPMs showed significant limitations when measured against the SVSM-vTPM's security properties. Most notably, they were not backed by hardware and implemented in the hypervisor. This meant that you still have to put a large amount of trust in the cloud provider. Although both platforms allow you to generate hardware attestation reports, this only assures that a system is booted in a trusted state.

Based on the comparison in Table 3.1, Azure's vTPM was identified as the most similar alternative to the SVSM-vTPM, satisfying 5 out of the 7 security properties. The primary limitations of Azure's design are its statefulness and closed-source nature. A stateful vTPM design may introduce risks such as state manipulation or rollback attacks, while the lack of source transparency raises concerns about the implementation being insecure. However, it is worth noting that a stateful design enables key sealing and persistent identity, which can be valuable use cases. Moreover, Azure has announced intentions to open-source its HCL (which includes the vTPM), which can improve the transparency.

As Azure's vTPM was identified as the closest match to the SVSM-vTPM, this work will use this result to implement secure remote attestation on Azure's vTPM in Chapter 4, aiming to provide users with the strongest achievable trust guarantees within current public cloud constraints.

It is also important to note the limitations of this analysis. Due to not having access to these platforms, the comparison was based solely on publicly available documentation, such as official specifications, frequently asked questions, and technical discussions from community forums and platforms like GitHub. This means that some capabilities or limitations may have been misinterpreted.

Chapter 4

Implementing Secure Remote Attestation with Keylime

This chapter presents the process of implementing secure remote attestation using the Keylime framework. Initially, a containerized approach was pursued, which revealed several challenges. Because of limited progress and community feedback, the strategy was revised, leading to a more effective incremental approach. This chapter outlines the technical journey, highlights key decisions, and shares valuable insights and tools for someone trying to implement secure remote attestation themselves. Finally, building on the findings from Chapter 3, an attempt is made to implement secure remote attestation using Azure's vTPM on CVMs.

4.1 Initial Approach: Containerized Keylime

4.1.1 Motivation for Containerization

Keylime, as discussed in Section 2.4.5, is a multicomponent framework that consists of the Verifier, Agent, Registrar, and Tenant, all of which interact over the network using mTLS. Each of these components has its own configuration and role within the remote attestation workflow. Furthermore, Keylime has significant dependencies, particularly related to cryptography and interaction with TPMs.

To manage this complexity and maintain isolation between services, using containerization looked like a good strategy. This is because Docker containers package all used dependencies neatly, and allow you to build a reproducible isolated environment for the component, independent of the Host OS. Additionally, they are quick to deploy and easy to manage. They also enable experimentation in a controlled, networked setup without modifying the host system, except for installing Docker. Additionally, using containers can offer benefits concerning scalability, which could

be useful when deploying remote attestation at scale [64].

4.1.2 Setup Overview

The initial implementation strategy involved deploying each Keylime component in separate Docker containers to emulate a distributed environment. A software-based TPM emulator (swtpm) was also containerized, with the Agent configured to interact with it. This setup facilitated the simulation of remote attestation within a controlled, isolated environment.

The plan included distributing the necessary certificates among the components to establish mutual TLS (mTLS) connections, configuring the components, and deploying Keylime workflows like remote attestation and secret sharing. Utilizing Docker Compose, the goal was to orchestrate the multi-container setup locally and efficiently (see Figure 4.1). Docker Compose is a tool that allows for defining and managing multi-container Docker applications through a YAML configuration file. This file specifies the services, networks, and volumes required, enabling straightforward deployment and management of containers using simple commands [65].

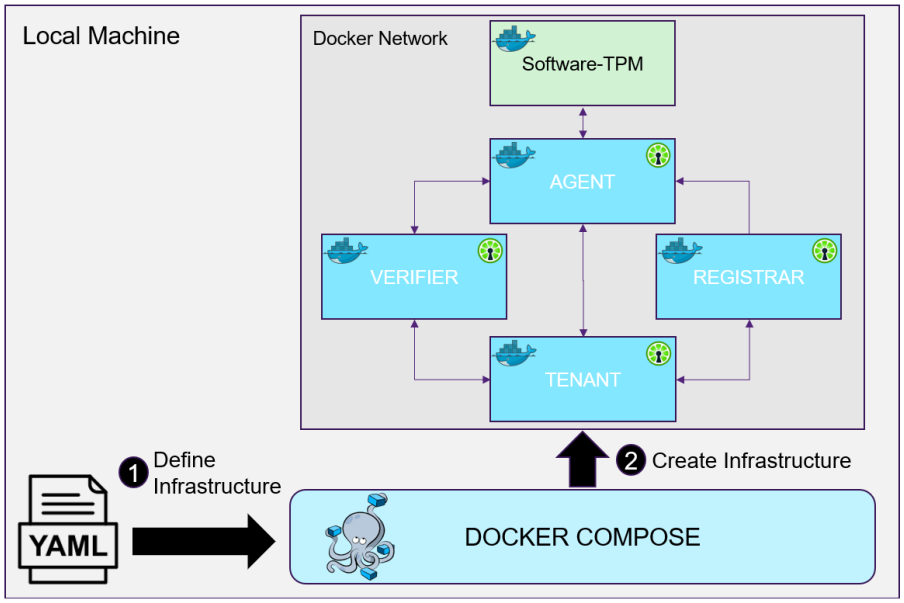


Figure 4.1: Setting up Keylime environment with Docker Compose

Upon achieving a functional prototype in this containerized environment, the subsequent phase aimed to transition the deployment to cloud infrastructure equipped with a Azure's vTPM. This progression was intended to validate the implementation in a more realistic setting. Considerations were made to explore deployment in public cloud environments using Docker Swarm for container orchestration, although

detailed planning for this phase was deferred to focus on immediate development milestones and faced challanges.

4.1.3 Challenges Encountered

Non-functional existing docker-compose file

To bootstrap some progress towards a working Docker environment, an old docker-compose file (`docker-compose.yml`) was found in the Keylime's Rust repository [66]. This setup used a swtpm, that was created on the container of the Agent. The Dockerfiles used by this setup were very outdated: additional dependencies needed to be installed, outdated commands had to be updated, and certain references to no longer existing files had to be removed. Furthermore, the Dockerfiles included warnings indicating that they were intended only for automated testing purposes and not for general use, as shown in Figure 4.2. Due to these limitations, it was decided to create a new docker-compose file using updated base images and configurations.

```
################################################################################
# keylime TPM 2.0 Rust Dockerfile
#
# This file is for automatic test running of Keylime and rust-keylime.
# It is not recommended for use beyond testing scenarios.
################################################################################
```

Figure 4.2: Warning usage Dockerfile (/docker/fedora/keylime_rust.Dockerfile)

Trying to fix this docker-compose file, did give initial insight into Keylime. It showed on which ports the components listen by default (not specified in official documentation):

- **Registrar**: ports `8890` and `8891` (unsecure and secure communication) .
- **Verifier**: port `8892`.
- **Agent**: port `9002` .

Additionally, it was learned that the Verifier creates a bunch of server and client certificates signed by newly create root certificate. These are generated in `/var/lib/keylime/cv_ca` when a user does not specify its own certificates, and are used to set up the mTLS communication. Lastly, we learned that the components are configured to run with root privileges, as indicated by the `user:root` directive in the `docker-compose.yml` file. This is because these components require access to low-level resources like TPM devices, mound secure filesystems, and bind to privileged network ports.

Image Creation

To create our own docker-compose file, working images of the components were needed. To create these, a script (`/docker/release/build_locally.sh`) provided by Keylime was found and used.

The first issue arose from running the script on Windows, the operating system used for this thesis. The script was designed for Linux, so it failed to execute properly. To overcome this, Windows Subsystem for Linux (WSL) was used. On WSL, additional dependencies such as `skopeo` and `jq` had to be installed for the script to run successfully.

This script, located in the Python repository, did not build the Agent image, as the agent was part of the Rust repository. Fortunately, another `build_locally.sh` script was found in the Rust repository. However, this script encountered issues with the Cargo lock file (see Figure 4.3). Multiple troubleshooting steps were taken, such as modifying the lock file version, updating Cargo, installing additional dependencies, and adjusting the `PKG_CONFIG_PATH` variable, but the issue persisted.

```
 > [builder 12/14] RUN make RELEASE=1 TARGETDIR=target target/release/keylime_agent:
0.398 Package systemd was not found in the pkg-config search path.
0.398 Perhaps you should add the directory containing `systemd.pc'
0.398 to the PKG_CONFIG_PATH environment variable
0.398 No package 'systemd' found
0.400 cargo build --target-dir="target" --release
0.987 error: failed to parse lock file at: /src/rust-keylime/Cargo.lock
0.987
0.987 Caused by:
0.987   lock file version `4` was found, but this version of Cargo does not understand this lock file,
perhaps Cargo needs to be updated?
0.990 make: *** [GNUmakefile:26: target/release/keylime_agent] Error 101
```

Figure 4.3: Docker image for Rust Agent not building.

The problem was ultimately resolved by setting the image's distribution type to Fedora. This was achieved by defining the `DOCKERFILE_TYPE=fedora` environment variable before executing the script.

Setting Up and Connecting to a Swtpm

During the early stages of the setup, one of the more complex tasks was integrating a swtpm into the containerized Keylime environment. Initially, the swtpm was run in the container together with the agent, to mimic to method used in the old docker-compose file.

To keep the new docker-compose file clean, the swtpm initialization logic was moved into a separate `entrypoint.sh` script (available here). For this setup to function correctly, several additional dependencies had to be installed in the container:

`dbus-daemon`, `swtpm`, `swtpm-tools`, and `tpm2-abrmd`. The script was in charge of initializing a virtual TPM state directory using `swtpm_setup`, launching the swtpm socket-based emulator, starting the D-Bus system service (used for inter-process communication), and configuring the TPM2 Access Broker, which coordinates access to the TPM among multiple applications. Multiple iterations were required to get the setup working, due to issues related to missing dependencies, system permissions, and the correct startup sequence of services.

Once the Agent container was running with the swtpm, the Keylime Agent produced errors claiming that it could not connect to the swtpm (see Figure 4.4). The error occurred because the TPM Command Transmission Interface (TCTI), which defines how the TPM2 software stack communicates with a TPM device or emulator, was not correctly configured via the environment variable. Initially, the TCTI was set to mssim, as recommended in Keylime's documentation [11], but this failed. This led to the assumption that the swtpm instance was not correctly initialized. Several approaches were explored, including running swtpm in the same container as the agent, in a separate container, and directly on the host machine, none of which resolved the issue.

```
keylime_agent-1      | WARNING:tcti:src/tss2-tcti/tcti-mssim.c:478:simulator_setup()
Failed to send MS_SIM_NV_ON platform command.
keylime_agent-1      | ERROR:tcti:src/tss2-tcti/tctildr-dl.c:149:tcti_from_file()
Could not initialize TCTI file: mssim
keylime_agent-1      | ERROR:tcti:src/tss2-tcti/tctildr.c:477:tctildr_init_context_data()
Failed to instantiate TCTI
keylime_agent-1      |
keylime_agent-1      | thread 'main' panicked at keylime/src/tpm.rs:515:67:
keylime_agent-1      | Failed to create TPM context: TSSTctiContextError
```

Figure 4.4: Agent not connecting to swtpm

To support debugging and allow supervisors to reproduce the setup, Vagrant was used to provision a consistent virtual machine environment. Vagrant is a tool for building and managing reproducible development environments using virtual machines. It allows defining the setup in a single configuration file (Vagrantfile), making it easy to share and recreate identical environments across different systems [67]. The Vagrantfile that was created and used for this debug process can be found here.

Eventually, the TCTI was altered with `TCTI=swtpm:host=swtpm,port=2321`, which allowed the TPM context to finally initialize correctly. However, the Keylime Agent then produced a new set of errors related to permission issues, which, despite multiple attempts, could not be fully resolved.

This challenge provided valuable insight into how to create and connect to a swtpm, how to use the TCTI variable, and how to leverage Vagrant to set up reproducible environments for experimentation and troubleshooting.

4.2 Rethinking the Approach

Despite extensive efforts to containerize the Keylime Agent, progress remained slow and yielded limited insight into configuring and using Keylime effectively. With the thesis limited to a 13-week timeframe, it became clear that continuing this direction risked leaving insufficient time to demonstrate meaningful results. Additionally, repeated issues and a cascade of new errors using the container-based approach made development inefficient and unpredictable.

Further motivation to change direction came from discussions with the Keylime developer community through their available Slack channel. When inquiring about the feasibility of containerizing the Agent, it was confirmed that this setup is known to be particularly challenging due to the Agent's need for low-level access to system resources. They explained that because of this complexity, no one had been working on deploying the agent in containers.

Support for changing approach was also encouraged by external expertise. A call was arranged with Jordi Thijsman, a PhD student at UGent with expertise in remote attestation and experience using Keylime. Based on the situation, his advice was to abandon the containerization strategy in favor of a more incremental approach.

4.3 Incremental Approach Using Virtual Machines

4.3.1 Motivation

As discussed in the previous section, the container-based approach faced numerous challenges and slow progress. The new approach focuses on installing Keylime components directly on virtual machines, which is more straightforward since most Keylime documentation and community examples revolve around VM deployments. Using VMs also offers greater control over the environment, allowing for easier debugging and configuration adjustments. Furthermore, Keylime provides existing tools and Ansible roles designed for provisioning on VM hosts, including utilities to launch a TPM emulator and an IMA emulator. This setup offered an immediate, practical way to start experimenting with the framework and gain a deeper understanding of its operation.

4.3.2 Overview

This section presents an incremental methodology for deploying and configuring Keylime to build a realistic attestation environment. It directly supports the second goal of this thesis: deploying secure, independent remote attestation using Keylime.

By progressively introducing more realistic and complex setups, this methodology enables both technical validation and improved documentation of the framework.

- **Single-VM Setup with Keylime and TPM emulator:** The initial step involves installing and configuring all Keylime components on a single virtual machine provisioned by Vagrant, alongside a TPM emulator and an IMA emulator. Afterwards, some key functionalities of Keylime will be tested and documented. This foundational setup helps to understand Keylime's architecture and establish a working baseline.

- **Multi-VM Keylime setup:** Next, the deployment is extended to multiple virtual machines using Vagrant, distributing the components across hosts. This introduces additional complexity, such as certificate management for mutual TLS (mTLS) and network configuration beyond localhost, simulating a more realistic scenario.

- **Automating Keylime with Ansible:** To streamline the deployment and reduce manual effort done in the previous steps, Ansible playbooks are developed to automate Keylime's configuration. This improves reproducibility and facilitates easier environment provisioning and configuration. These automation scripts include support for real TPM hardware. This playbook is extensively tested on the multi-VM setup from the previous step.

- **Creating cloud infrastructure on Azure:** As part of the effort to implement secure remote attestation using Azure's vTPM, this step involves provisioning a full multi-VM setup in the cloud, similar to the earlier local environment. Using Pulumi as the infrastructure-as-code (IaC) tool, multiple Azure virtual machines are deployed and connected over a private network. One of these is a CVM intended to host the agent, to explore Azure's vTPM capabilities. This automated approach ensures reproducibility and provides a cloud environment to further experiment with Keylime.

- **Secure remote attestation using Azure's vTPM:** With the cloud-based multi-VM infrastructure in place, this step aims to achieve fully independent remote attestation using Azure's virtual TPM. Extensive experiments are conducted to evaluate vTPM integration with Keylime, leading to the development of a dedicated Ansible playbook. This final stage demonstrates the feasibility of secure, vendor-independent attestation on public cloud infrastructure.

4.3.3 Single-VM Setup with Keylime and TPM emulator

To build a foundational understanding of Keylime's components and operations, the initial phase involved testing Keylime with the keylime-vagrant-ansible-tpm-emulator repository [32]. This is a repository with a Vagrant file to create a virtual machine, and Ansible roles to provision the VM with Keylime's components, a TPM emulator, and an IMA emulator. Ansible is an open-source automation tool that simplifies the process of configuring and managing servers, applications, and IT infrastructures. It operates by connecting over SSH from a central control node to target machines, executing instructions defined in YAML-based playbooks . The use of these tools facilitated a rapid setup, allowing for immediate engagement with Keylime's features.

Before initiating the setup, it was necessary to install Vagrant and a virtualization provider, such as VirtualBox. Vagrant interacts with VirtualBox to create and configure the VM locally as specified in the Vagrantfile. By executing the command `vagrant up`, Vagrant communicates with VirtualBox to instantiate and configure the VM.

To make the emulators work, the VM was rebooted using `vagrant reload`, followed by restarting the TPM emulator and IMA emulator services in that order. This activated the basic ima-policy put in `/etc/ima/ima-policy` by the Ansible playbook. An IMA policy instructs the IMA, discussed in Section 2.4.4, what files it should measure for integrity. In our setup, the provided policy (see Figure 4.5) excludes certain virtual filesystems (e.g., `/proc`,`/sys`) from measurement and focuses on monitoring executed binaries (`BPRM_CHECK`), files mapped into memory with execute permissions (`FILE_MMAP`), and kernel modules loaded by the root user (`MODULE_CHECK` with `uid=0`) [68]. It serves as a starting point and should be adapted to match specific security requirements in real-world deployments.

```
# PROC_SUPER_MAGIC
dont_measure fsmagic=0x9fa0
# SYSFS_MAGIC
dont_measure fsmagic=0x62656572
# DEBUGFS_MAGIC
dont_measure fsmagic=0x64626720
# TMPFS_MAGIC
dont_measure fsmagic=0x01021994
# RAMFS_MAGIC
dont_measure fsmagic=0x858458f6
# SECURITYFS_MAGIC
dont_measure fsmagic=0x73636673
# MEASUREMENTS
measure func=BPRM_CHECK
measure func=FILE_MMAP mask=MAY_EXEC
measure func=MODULE_CHECK uid=0
```

Figure 4.5: Basic IMA policy used in setup

All Keylime components were started manually using the `keylime_<component_name>` command as root in separate terminal sessions. It was essential to start the registrar before the agent, as the agent attempts to register itself on startup and will fail if the registrar is not yet available. Since all components were running on the same machine, no configuration changes were necessary because Keylime's default settings are designed to support local operation. Additionally, the verifier automatically generates the mTLS certificates needed for secure communication when no certificates are specified in the configuration, and because all components share the same environment, these certificates are readily accessible without the need for distribution.

The agent initially encountered issues connecting to the TPM emulator. This was resolved by setting the environment variable `TPM2TOOLS_TCTI` to `"device:/dev/tpm0"`, contrary to the repository's recommendation to use `"mssim:port=2321"`. This solution worked because the TPM emulator was initialized as a character device, which interfaces through `/dev/tpm0` rather than network sockets. Furthermore, the agent lacked permission to access `/var/lib/keylime` and was solved by executing:

```
chown -R keylime:tss /var/lib/keylime
```

This solution was discovered by looking through the agent configuration files, where it is documented that the agent drops root privileges, and that command needed to be executed. The reason why the agent drops its privileges is to improve security and is further discussed in [69].

To ensure that quotes from the emulated TPM are trusted, its EC issuer certificate was added to Keylime's certificate store at `/var/lib/keylime/tpm_cert_store`. This directory holds certificates from trusted TPM manufacturers and is used by the tenant to verify whether a TPM can be trusted during the attestation process. The issuer certificate is the public certificate of the authority that issued the TPM's EC. It allows the tenant to verify the signature on the EC and confirm its authenticity. In the case of the emulated TPM, this issuer certificate was found at `/var/lib/swtpm-localca/issuercert.pem` and was manually copied into the Keylime certificate store.

Now that everything is set up, we need to create a policy to deploy on the agent. This was done by running:

```
create_runtime_policy.sh  -o test_runtime_policy.json -a sha256sum
-e excludes_list.txt
```

This script scans your system to record the current state of important files. It creates a "baseline" file (`test_runtime_policy.json`) that lists these files and their checksums, serving as a reference to detect any future unauthorized changes. The `excludes_list.txt` file specifies which files or directories to ignore during this process, typically those that change frequently or are not critical to system integrity [11]. This policy was deployed on the agent using the following command:

```
keylime_tenant -v 127.0.0.1 -t 127.0.0.1 -u <UUID>
--runtime-policy test_runtime_policy.json
```

The -v and `-t` options specify the verifier and agent IP addresses, respectively. The `-u` option sets the agent's UUID, which can be defined in `/etc/keylime/agent.conf`. This command successfully starts the attestation process, discussed in section 2.4.5, where the verifier periodically polls the agent status and verifies it.

To test whether attestation was functioning correctly, a simple script named `evil.sh` was created and executed. This was intended to simulate an attacker who had gained root access to the VM and attempted to compromise the system. Although the script merely contained an `echo` command, this was sufficient to trigger a revocation of the agent by the verifier, as seen in Figure 4.6. This occurred because the execution of `evil.sh` was not excluded by the runtime policy, allowing it to be checked and flagged as an unexpected change. This confirmed that attestation was working as intended.

```
2025-06-06 06:50:33.179 - keylime.tpm - INFO - Checking IMA measurement list on agent: d432fbb3-
d2f1-4a97-9ef7-75bd81c00000
2025-06-06 06:50:33.179 - keylime.ima - WARNING - File not found in allowlist: /root/evil.sh
2025-06-06 06:50:33.179 - keylime.ima - ERROR - IMA ERRORS: Some entries couldn't be validated.
Number of failures in modes: ImaNg 1.
2025-06-06 06:50:33.265 - keylime.verifier - WARNING - Agent d432fbb3-d2f1-4a97-9ef7-75bd81c0000
0 failed, stopping polling
```

Figure 4.6: Output on Keylime's Verifier when executing an untrusted script.

4.3.4 Multi-VM Keylime setup

Now that attestation is working on a single VM, the complexity and realism of the setup are increased by deploying each component on a separate VM. This step uses a custom repository created to enable scenario customization and support the following phases of the thesis. The repository is available at:
https://github.com/JelleDeMoerloose/thesis_keylime_deployment
Ansible roles from the previous step — used to provision Keylime and install the emulators — were also included in this repository repository and later adapted.

A Vagrantfile was created to provision four separate Fedora-based virtual machines, each representing a Keylime component: agent, registrar, verifier, and tenant. Fe-

dora was chosen as the base image because the official Keylime Ansible provisioning roles are designed specifically for Fedora systems. Each VM was assigned a static private IP address to allow consistent and predictable communication between components. However, the default Vagrant network configuration did not correctly bind the IP addresses to the secondary network interface (`enp0s8`), as confirmed by inspecting the output of `ip -a`. To address this, `nmcli` commands were used during provisioning to explicitly add and activate the connections on the correct interface, ensuring proper network configuration. The provisioning script for the tenant VM additionally installs Ansible and copies the Vagrant-generated SSH private keys of all other VMs into a dedicated .ssh directory. This configuration allows the tenant VM to function as a central control node for executing the Ansible playbooks on all the components. This approach was necessary due to the limitations of running Ansible directly on Windows, which does not natively support it.

After the machines were created, the repository was cloned on the tenant and a host file was created, as shown in Figure 4.7. This file specifies the IP addresses of each component, matching those defined in the Vagrantfile, along with the SSH keys used for authentication. It is utilized during the execution of an Ansible playbook to determine which instructions are applied to specific groups of hosts.

```
[keylime_agent]
keylime_agent ansible_host=192.168.33.10 ansible_user=vagrant ansible_ssh_private_key_file=~/.ssh/keylime/keylime_agent_key

[keylime_registrar]
keylime_registrar ansible_host=192.168.33.11 ansible_user=vagrant ansible_ssh_private_key_file=~/.ssh/keylime/keylime_registrar_key

[keylime_verifier]
keylime_verifier ansible_host=192.168.33.12 ansible_user=vagrant ansible_ssh_private_key_file=~/.ssh/keylime/keylime_verifier_key

[keylime_tenant]
keylime_tenant ansible_host=192.168.33.13 ansible_user=vagrant ansible_ssh_private_key_file=~/.ssh/keylime/keylime_tenant_key
```

Figure 4.7: Used hosts_vagrant.ini file for Ansible playbook

To provision all the components with Keylime and the agent with the emulators, the following command was successfully executed on the tenant:

```
ansible-playbook -i hosts_vagrant.ini playbook.yml
```

After provisioning the VMs, it was essential to configure them appropriately and distribute the necessary certificates. Keylime employs mTLS for most inter-component communications, ensuring both authentication and confidentiality. In mTLS, the client initiates the connection, and both client and server authenticate each other's certificates, which are signed by a trusted CA. To facilitate these interactions, all components require appropriate certificates: server certificate and key for authenticating themselves to clients, client certificate and key for authenticating themselves to servers, and the CA certificate to verify the authenticity of the certificates presented by other components. By default, the verifier generates the CA and associ-

ated certificates upon startup, storing them in `/var/lib/keylime/cv_ca/`. These certificates were then securely copied using `scp` to the respective components based on their roles (client, server, or both), as depicted in Figure 4.8, which shows the client-server relationship among the Keylime components [70]. Notably, the agent registers with the registrar over plaintext HTTP by default. This design choice is intentional, as the registration protocol, discussed in Section 2.4.5, leverages TPM-based cryptographic operations to establish trust.

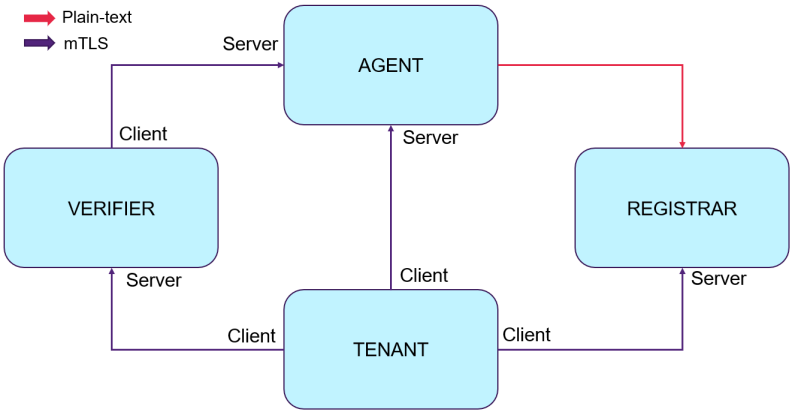


Figure 4.8: Keylime communication relationships between components

To configure the Keylime components, environment variables were used. This approach offers flexibility and clarity, allowing for straightforward adjustments of default configurations. Each component's environment variables were set to specify parameters such as IP addresses, ports, and paths to TLS certificates and keys. To ensure these configurations persist across system reboots and are automatically applied in new shell sessions, the export statements were appended to the `root/.bashrc` file. For example, the following lines set the necessary environment variables for the Keylime Agent in the multi-VM setup:

```
export KEYLIME_AGENT_IP="192.168.33.10"
export KEYLIME_AGENT_PORT="9002"
export KEYLIME_AGENT_REGISTRAR_IP="192.168.33.11"
export KEYLIME_AGENT_REGISTRAR_PORT="8890"
export KEYLIME_AGENT_SERVER_KEY="server-private.pem"
export KEYLIME_AGENT_SERVER_CERT="server-cert.crt"
export KEYLIME_AGENT_TRUSTED_CLIENT_CA="cv_ca/cacert.crt"
```

This process was repeated for each Keylime component, with the respective environment variables tailored to their specific roles.

Furthermore, the issuer certificate from the swtpm on the Agent VM had to be copied into `/var/lib/keylime/tpm_cert_store` on the tenant VM, as the tenant is

in charge of checking the EC provided by the agent.

Lastly, the steps described in Section 4.3.3 were replicated to start remote attestation, including the scenario where the agent is expected to fail attestation. Thanks to the correct configuration of environment variables and the proper distribution of certificates, the process executed successfully in the multi-VM environment.

4.3.5 Automating Keylime with Ansible

To address Keylime's tedious and sparsely documented setup process, this step aims to automate the deployment using an Ansible playbook. The playbook was developed based on the insights and configurations established in the previous two steps and can be found in the repository used in the last step.

Ansible was selected as the configuration tool because of its modular and declarative design. Additionally, Keylime has already released Ansible provisioning roles targeting Fedora operating systems, and can be reused and extended. Alternatives such as embedding setup logic directly in provisioning tools like Vagrant, or using Infrastructure-as-Code frameworks such as Terraform or Pulumi, were deliberately avoided. These approaches tend to couple configuration tightly with infrastructure, violating the principle of separation of concerns. Using Ansible allows configuration to remain decoupled and reusable across different environments and infrastructure, improving both portability and maintainability.

The `playbook.yml` file developed for this project consists of multiple Ansible plays, which are illustrated in Figure 4.9. A play is a set of tasks or roles that are executed on a defined group of hosts. In the initial play, all hosts are provisioned with Keylime using the pre-existing Ansible role. This step is executed in parallel across hosts, as it is relatively time-consuming and benefits from concurrent execution. Subsequently, each Keylime component is individually configured in separate plays. This modular approach is necessary because the configuration steps and certificate requirements vary across components. During this stage, environment variables are defined and injected, and the relevant certificates are placed in their designated locations, based on our findings from the previous step. In the case of the agent, an additional configuration step is required to assign proper file and directory permissions, as discussed in Section 4.3.3 . To streamline certificate distribution, a custom role `keylime_mTLS_setup` was created. It standardizes the distribution of certificates and reduces duplication across plays, improving modularity and readability of the playbook. Optionally, the agent can be provisioned with a TPM and IMA emulator, usefull for experimenting in VMs without TPM access. For this to work, the tenant is provided with swtpm issuer certificate, as explained in Section 4.3.4.

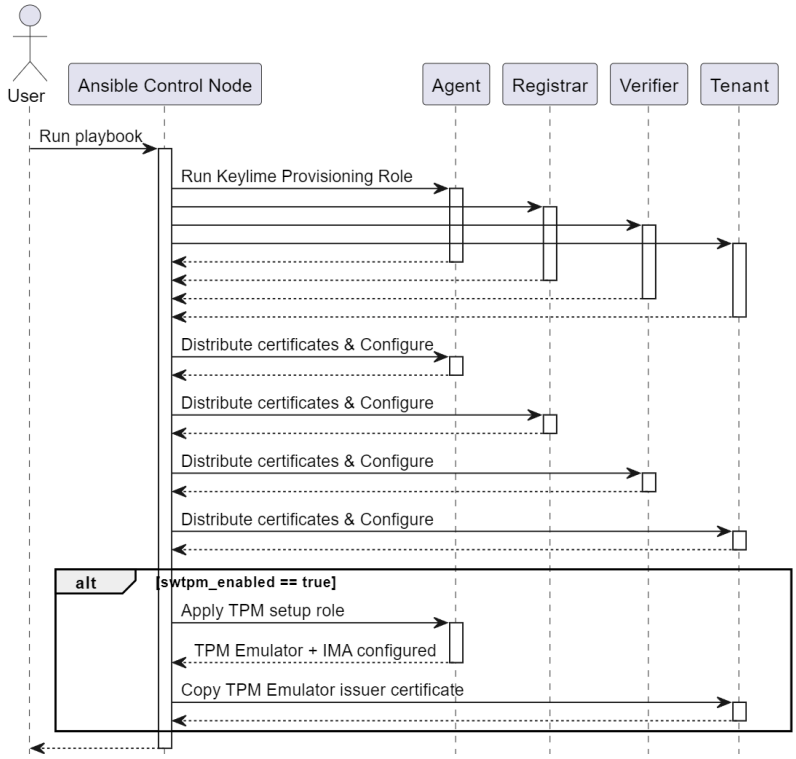


Figure 4.9: Sequence diagram of the working of the developed Ansible playbook

To improve usability and flexibility, the playbook was modified to use variables. This way the playbook can be customized to be used in different deployment scenarios. For example, IP addresses, port numbers, and file paths are all definable inside of the `host_vars` directory. Global variables like the use of an emulated TPM are defined in the `group_vars` directory. Furthermore, the playbook automatically uses the provided IP-addresses defined `hosts.ini`, making it even easier to use.

To facilitate the use of this automation tool in Azure (needed in later steps), the existing Keylime provisioning role was extended to support Ubuntu 22.04+, in addition to the original support for Fedora 35+. This was necessary because Azure's confidential VMs (needed in next steps) only officially provide Ubuntu images, with no supported or available Fedora options. To enable cross-platform compatibility, OS-specific tasks were conditionally included using `ansible_facts['distribution']`, which can detect the OS of the host where the role is executed. The created extension `ubuntu_install.yml`, installs the equivalent dependencies needed for Keylime to work on Ubuntu and downloads some unavailable ones from Github. Additionally, the extension handles the creation of missing system users (`tss`, `keylime`) and initializes the required directory structure with correct permissions.

The playbook was tested extensively using the VM setup from the previous step, described in Section 4.3.4. This process helped identify and correct issues in the

playbook itself during development.

4.3.6 Creating Cloud Infrastructure on Azure

In this step, Azure cloud infrastructure is provisioned using Infrastructure-as-Code (IaC) to establish an environment for experimenting with independent remote attestation using Azure's vTPM in the following sections.

Infrastructure-as-Code is a methodology for managing and provisioning computing infrastructure (.e.g. VMs, networks, loadbalancers) through code, rather than manual processes. It enables automated, consistent, and repeatable deployments, which is especially useful in complex or cloud-based environments.

For this project, Pulumi was chosen as the IaC-tool due to its support for writing infrastructure definitions in general-purpose programming languages such as TypeScript. In contrast, Terraform was not selected because it relies on HashiCorp Configuration Language, which would have introduced a learning curve that was not practical given the limited time frame of the project [71].

First, Pulumi was installed together with the Azure CLI. By logging into the Azure account with `az login`, Pulumi can automatically access your subscription to start creating infrastructure. To speed up development, a Pulumi template project was created in the `/IaC` directory of the repository, using:

```
pulumi new vm-azure-typescript
```

This command creates an `index.ts` file, that defines the resources required to deploy a VM on Azure. As shown in Figure 4.10, it provisions a virtual network with a subnet and a network security group, which acts like a firewall by controlling allowed inbound and outbound traffic. It also creates a network interface with both a private and a public IP address to enable internal and external connectivity. It creates and provisions the virtual machine, attaches the network interface, and specifies the VM size and OS image. To enable access of the VM via SSH, an SSH-keypair is created and the public part is added to the `/.ssh/authorized_keys` folder of the VM.

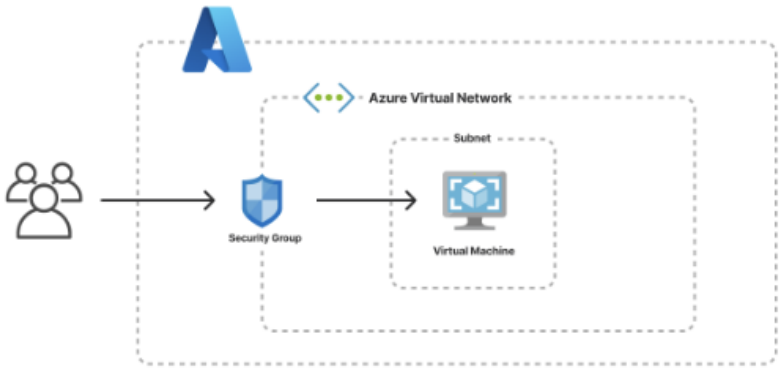


Figure 4.10: Infrastructure defined by Pulumi template [72]

This template was altered to create a multi-VM setup, where one VM gets created per Keylime component and gets connected to that virtual network using a loop. The tenant, verifier, and registrar are all deployed using the `Standard_B2s` VM size, a cost-effective, general-purpose instance type with sufficient memory to support the use of the developed Ansible playbooks.

For the agent, a CVM was created using the `Standard_DC2as_v5` VM size, as this enables access to AMD SEV SNP hardware and Azure's vTPM. However, to successfully deploy this VM size, it was necessary to specify Availability Zone 2 within the East US region, as this particular zone supports the that VM size. Additionally, the VM's security profile was configured by setting the `securityType` to `ConfidentialVM` and enabling UEFI settings such as secure boot and vTPM. Furthermore, the OS disk was configured with the `securityEncryptionType` as `DiskWithVMGuestState`, as Azure's forces you to use encrypted disks when using CVMs to ensure integrity and confidentiality of data at rest. Lastly, to leverage Azure's hardware-based security features on this VM, Canonical's Ubuntu 22.04 LTS Confidential VM image was selected for its support of confidential computing.

To facilitate easy remote access to these VMs, a single SSH key was shared among all VMS. Code was added to the `index.ts` file to write the private part of this key to a file for convenient use. Furthermore, this key can be used by the Ansible control node to execute the playbooks, created in Section 4.3.5, on the infrastructure.

By running `pulumi up` in the `/IaC` directory, Pulumi starts creating the defined infrastructure, like shown in Figure 4.11. The infrastructure was verified for existence in the online Azure portal, seen in Figure 4.12, where you can see the created VMs.

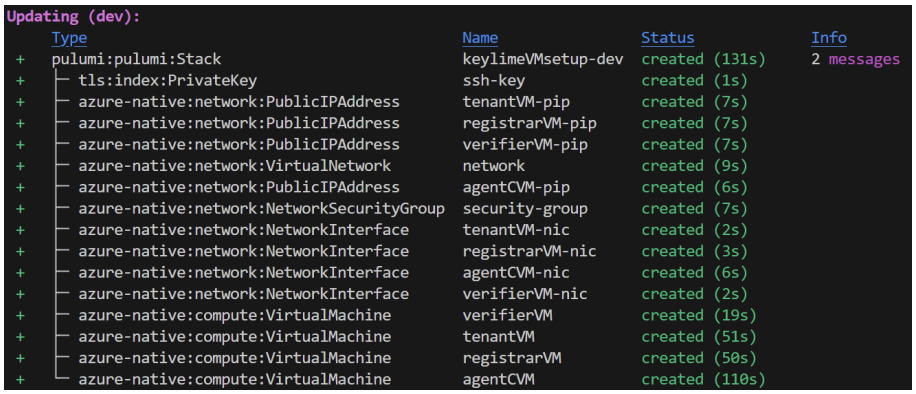


Figure 4.11: Created Keylime infrastructure using Pulumi

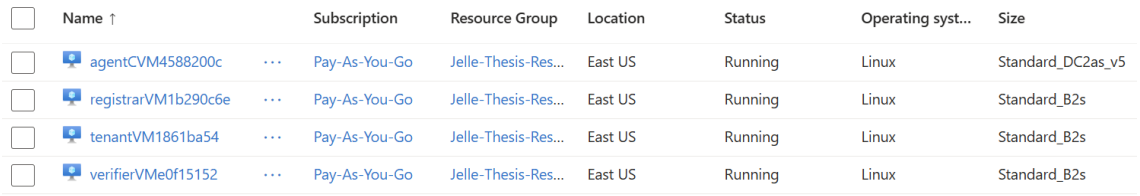


Figure 4.12: Created VMs displayed in Azure Portal

4.3.7 Secure remote attestation using Azure's vTPM

Now that the public cloud infrastructure is created, this step aims to implement independent remote attestation with Azure's vTPM, using all the knowledge and tools created in previous steps. First, Azure's vTPM was thoroughly tested, and based on the findings, Keylime's provisioning and configuration was tailored to Azure's CVMs by creating an additional Ansible playbook.

Analyzing Azure's vTPM

After accessing the Azure CVM (created in the previous step) via SSH, `tpm2-tss` and `tpm2-tools` were installed to inspect the Azure vTPM. First, the NV-indexes and handles were checked to see what certificates were stored:

```
root@agentCVM:~# tpm2_getcap handles-nv-index
- 0x1400001
- 0x1C101D0
root@agentCVMVM:~# tpm2_getcap handles-persistent
- 0x81000001
- 0x81000003
- 0x81010001
```

Notably, the standard EC indices (`0x01C00002` for RSA and `0x01C0000A` for ECC) are absent, indicating that the EC is not provisioned in this environment. This is problematic because the registration protocol of Keylime, discussed in Section 2.4.5, relies on the presence of the EC to trust an Agent. By inspecting these nv-indexes using `tpm2_readpublic` and looking at documentation in [73, 74], the meaning of these values were deducted and displayed in Table 4.1.

Table 4.1: List of Azure's vTPM populated NV-indices and their descriptions

Name	**NV-index**	**Description**
Attestation Report	0x1400001	Azure-defined format with the SEV SNP hardware report embedded.
AK certificate	0x1C101D0	Attestation Key certificate
SRK public	0x81000001	Used for sealing mechanisms
AK public	0x81000003	Used to verify quotes generated with sAK private
EK public	0x81010001	Used to identify and authenticate the TPM

The Azure attestation report, found in the NV-indexes, is a structured binary file composed of three main sections: the Header, the Report Payload, and Runtime Data. The Header is a 32-byte segment at the beginning of the report, which provides metadata such as versioning (not relevant). The raw AMD SEV-SNP hardware reports follows immediately and spans 1184 bytes. The remaining bytes (Runtime Data) of the Azure attestation report are JSON and contain the runtime claims and the public portions of both the AK and EK. This Runtime Data is hashed and included in the user-provided data section of the raw SEV-SNP hardware report, essentially rooting the claims and keys to a hardware root of trust [73].

Manual verification of the Azure attestation report was conducted to gain a comprehensive understanding of the attestation process. First, the hardware report was extracted using:

```
% tpm2_nvread -C o 0x01400001 > ./snp_report.bin
% dd skip=32 bs=1 count=1184 if=./snp_report.bin of=./guest_report.bin
```

To verify this report was generated by real AMD hardware, the tutorial was followed detailed in [75], as tutorials provided by Azure were using outdated tools. Next, the keys in the Runtime Data section were verified to match the AK and EK found in the NV handles. Lastly, the Runtime Data was hashed using its specified hash algorithm and verified that it matches the `Report Data` field of the SEV SNP hardware report. The boot measurements of the VM, recorded in the SEV SNP report, could not be verified, as no ground truth values were found in Azure's documentation.

The AK certificate was also analyzed by printing it out using `openssl`. The `Issuer` field was set to `Global Virtual TPM CA - 01`, which is the certificate authority used by Azure to sign vTPM keys. After hours of searching, the certificate of this issuer was found in [76]. It was verified that the AK certificate was correctly signed by the issuer's certificate, establishing its authenticity. Furthermore, the AK was validated against the AK certificate, confirming its association with Azure (see Figure 4.13).

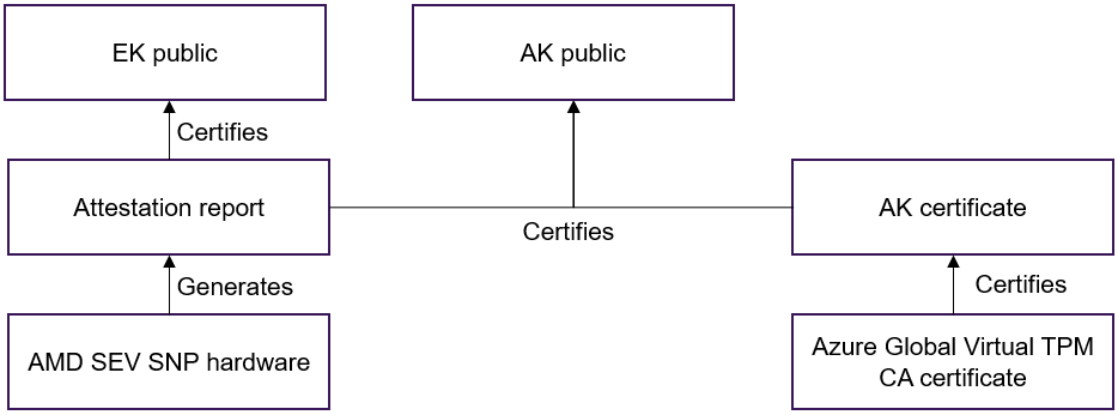


Figure 4.13: Certificate chain for verifying EK and AK of Azure's vTPM in CVM.

Implementing remote attestation with Keylime on Azure's vTPM

The analysis of Azure's vTPM revealed several adjustments needed to integrate it with Keylime. This is because Keylime uses an EC for the registration protocol, which is not available on the vTPM.

The best way to solve this is to provide the SEV-SNP attestation report, instead of the absent EC to authenticate the EK, just like in the SVSM-vTPM. But due to the limited timeframe of the project, that was not possible, as this meant large code changes had to be made to the agent, registrar, and tenant. Additionally, because this SEV-SNP report is embedded in the Azure report structure, the validation logic would be quite complex.

To still enable independent remote attestation with Keylime on Azure CVMs despite these limitations, the Initial AK (IAK) was used, found in `0x81000003`. This key is chosen because it includes a certificate signed by Azure, which can serve as a trust anchor during the registration process. This decision builds on Keylime's support for using the IAK and its accompanying certificate to register the agent with the registrar. This capability was identified by inspecting the registrar's configuration file (see Figure 4.14) and reviewing the limited available documentation [11].

```
# What TPM-based identity is allowed to be used to register agents.
# The options "default" and "iak_idevid" will only allow registration with IAK and IDevID if python cryptography is version 38.0.0 or higher.
# The following options are accepted:
# "default": either an EK or IAK and IDevID may be used. In the case that cryptography version is <38.0.0 only EK will be used
# "ek_cert_or_iak_idevid": this is equivalent to default
# "ek_cert": only allow agents to use an EK to register
# "iak_idevid": only allow agents with an IAK and IDevID to register
tpm_identity = "default"
```

Figure 4.14: Registration options configuration

To enable agent registration using the IAK, the `tpm_identity` parameter was set to `iak_idevid` in the `registrar.conf` file. This configuration directs the registrar to accept IAK credentials for device registration. To validate the certificates of the AIK, the registrar needs the issuer certificate, which was obtained in the previous section. This certificate must now be placed in the `/var/keylime/tpm_cert_store` directory so that the registrar can use it to verify the trustworthiness of the AIK. On the agent side, the IAK certificate was extracted from index `0x1C101D0` and placed in the path defined by the `iak_cert` configuration option. Additionally, the `aik_handle` was set to `0x81000003` so that Keylime knows where to find the IAK in the vTPM. The tenant, normally in charge of checking the EC, has to be configured with `require_ek_cert=false`, as there is no EC to check and the registrar already checks the IAK certificate (in in `registrar_agent.py`).[1]gkg

For a still unexplained reason, using this setup yielded the following error on the registrar:

```
ek_tpm must contain the same public key found in ekcert
```

This was puzzling, as no EC was present, a fact further confirmed by the agent, which reported the absence of an EC. To work around this issue, a customized version of Keylime was deployed on the registrar. In this version, the function causing the error (`_check_ek()` in `registrar_agent.py`) was placed in comments. This modification was deemed safe, as the function's sole purpose is to validate the structure and content of the EC, which is irrelevant in the current context due to its absence.

The steps required to configure Keylime for Azure's vTPM were also automated using a new Ansible playbook (`azure_playbook.yml`), enabling easy reproducibility and testing within the Azure cloud environment. After the cloud infrastructure was configured using the original `playbook.yml` (as described in Section 4.3.6), the `azure_playbook.yml` was executed to apply the Azure-specific configurations necessary for enabling remote attestation with Azure's vTPM.

[1]Keylime is gradually shifting certificate validation from the tenant to the registrar. Features like IAK-based registration already rely on registrar-side validation.

To validate the attestation functionality, the workflow outlined in Section 4.3.3 was re-executed using the properly configured Azure infrastructure. The process involved measuring the agent's current system state, generating a runtime policy from those measurements (with an excludes file), and deploying this policy with the tenant on the agent.

This successfully initiated independent remote attestation on the agent. However, executing the `evil.sh` script in `/root` did not trigger the verifier to revoke the agent. Investigation revealed that the CVM's IMA was configured to measure only the boot aggregate and kernel modules. This was determined by inspecting the IMA measurement log. To make sure the IMA of the CVM measures more events, a more extensive IMA-policy was copied into the `/sys/kernel/security/ima/policy`. Since this can only be done once per boot, the agent CVM cannot modify the policy at runtime. To enforce this, the policy update was integrated into the Pulumi infrastructure code (Section 4.3.6) using a small provisioning script that writes the policy into the CVM during boot. With the updated policy in place, the execution of `evil.sh` was detected and the agent was successfully revoked, demonstrating remote attestation using Azure's vTPM.

4.4 Final Result and Limitations

As a result of the stepwise progression outlined in this chapter, secure remote attestation was deployed on Azure's infrastructure with a confidential VM using Keylime (see Figure 4.15). This deployment directly supports the second project goal by demonstrating independent attestation with improved automation and documentation of Keylime.

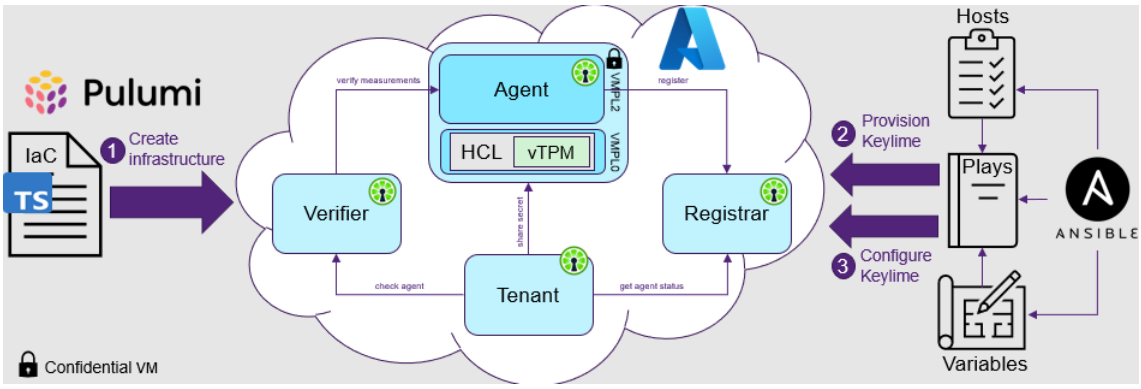


Figure 4.15: Final experimental setup for remote attestation using Azure's vTPM

The final infrastructure consists of four virtual machines on a virtual network, each made to host a distinct Keylime component, with the agent's machine configured as

a CVM to enable integration with Azure's vTPM. This infrastructure was created with Pulumi to make the cloud environment more reproducible and manageable. Keylime was deployed on these machines using the created Ansible playbooks, which automatically provision and configure the VMs, removing the need for a long and tedious setup process. This setup process, implemented using Ansible playbooks, can be adapted to custom environments by providing tailored variable and host files, making it more versatile and useful. Finally, remote attestation was executed on those machines and found to be working.

Despite achieving a functional remote attestation setup using Keylime, there are several important limitations to highlight:

- The current approach of using Ansible to provision virtual machines with Keylime works only on VMs running supported operating systems. While initial support has been extended beyond Fedora to include Ubuntu, further expansion to other OSes is still required. This limitation is inherent to VM-based provisioning, where the configuration depends heavily on the underlying OS. In contrast, a containerized approach avoids this issue entirely, as the operating system and application dependencies are encapsulated within the container image. This means that, as long as the Docker engine is available on a host, containers can run consistently across different environments regardless of the host OS. However, due limited development progress using the container-based approach and other reasons outlined in Section 4.2, the Ansible-based provisioning method has proven to be more successful.

- Currently, the trust guarantees of the final setup, as shown in Figure 4.15, are based on a certificate issued by Azure. While this provides a functional level of trust, it inherently requires relying on the cloud provider as a trusted third party. Ideally, trust could instead be rooted in the AMD-SEV SNP attestation report itself, which would eliminate the need to trust the cloud provider altogether. Due to limited time, opting for the Azure-issued certificate provided for a more practical and convenient path to a working solution.

- Beyond these implementation-specific limitations, a significant technological constraint of Azure's vTPM was identified. When manually performing the verification of the SEV-SNP report exposed by Azure's vTPM, the measurements included in the report could not be verified. This was because no documentation was found regarding the trusted reference values of this field. This is problematic because you cannot verify that VM (including the vTPM) is booted correclty.

Chapter 5

Conclusions

This project aimed to strengthen trust in public cloud computing through two main goals. The first goal was to evaluate whether the SVSM-vTPM , a design with very desirable security properties proposed by Narayanan et al. [7], could be deployed on current public cloud platforms, effectivly removing the need for trust in the cloud provider. While it had only been tested in an experimental environment, proving its feasibility in real-world settings would highlight its industry relevance. The second goal was to implement and improve the usability of the Keylime framework for independent remote attestation, as an alternative to opaque, provider-managed attestation mechanisms. Beyond its general use, Keylime was also intended to support remote attestation for the SVSM-vTPM, if a public cloud implementation proved possible.

To address the first goal, a comprehensive study was executed that validated a set of high-level deployment requirements against the three most popular cloud providers (AWS, Azure, GCP). These requirements were derived from an analysis of the design and Proof-of-Concept [12] of the SVSM-vTPM. The results showed that deploying the SVSM-vTPM on these platforms was not feasible, as none currently allow the cloud consumer to host custom services in VMPL0, a critical requirement. This finding underscores a critical gap between academic designs , such as the SVSM-vTPM [7], and real-world cloud capabilities, highlighting the need for public cloud platforms to evolve accordingly.

Although deploying the SVSM-vTPM was not feasible, its design remained central to the project. Rather than pursuing its direct implementation, the project shifted to identifying the most secure alternative currently available on public cloud platforms. The goal was to still provide the user with the strongest trust guarantees within the current offerings. To find the most secure alternative, a comparative analysis of cloud vTPMs was conducted where the security properties of the SVSM-vTPM's

were used as a benchmark. This analysis identified Azure's vTPM as the most similar, making it a strong candidate for the deployment of remote attestation. In doing so, the study also consolidated and clarified the often fragmented and hard-to-find documentation on the confidential computing capabilities of AWS, Azure, and GCP, offering a valuable resource for future research and secure cloud deployment decisions.

To address the second goal, an incremental approach was taken, gradually increasing in realism and complexity to build a solid understanding of the Keylime framework. The framework was thoroughly documented throughout, covering provisioning, configuration, policy creation, and attestation workflows. To further support adoption of Keylime and reduce entry barriers, Ansible playbooks were developed to automate the setup process and improve overall usability. This went beyond the original objective, transforming a fragmented and technically demanding process into a streamlined, reproducible deployment. Eventually, Keylime was implemented for secure remote attestation on a multi-VM Azure infrastructure using a confidential VM with Azure's vTPM. Due to time constraints, this implementation relied on Azure's certificates rather than hardware-rooted verification using SEV-SNP reports. Nonetheless, the setup demonstrates the viability of independent attestation in public cloud environments and lays the groundwork for future hardware-backed attestation. Reproducibility was further supported through infrastructure-as-code with Pulumi, ensuring the used Azure infrastructure can be easily replicated and modified. Together, these efforts contribute to reducing reliance on opaque attestation mechanisms and advancing the practical adoption of confidential computing.

As part of the second goal, Azure's vTPM was empirically analyzed to understand its capabilities and limitations for independent attestation. Unlike typical TPMs, it lacks an endorsement certificate because its key is hardware-rooted via the SEV-SNP report. Although the report was manually verified, limitations were identified, including the difficult validation of Azure's report format and the lack of trusted boot measurement values. Combined with the closed-source nature of the vTPM design, these findings highlight a critical gap that must be addressed by Azure to provide stronger trust guarantees.

Reflecting on the project execution, the original plan aimed to implement the SVSM-vTPM on public cloud infrastructure. This was ultimately not feasible due to limitations in current cloud platforms, leading to a creative pivot toward identifying and analyzing the most secure available alternative. For the second goal, the initial plan to implement Keylime through a containerized approach proved ineffective, largely due to recurring technical issues. More importantly, the plan itself was not suffi-

ciently thorough, making it difficult to approach the task systematically. This was mainly due to the limited planning time within the 13-week project and an initial lack of deep understanding of the complex field of confidential computing. Adapting to these challenges by shifting to a more incremental, VM-based approach was key to making progress and ultimately completing the implementation. In the final phase, only two weeks of access to Azure's platform were available, which made full integration of SEV-SNP–based attestation infeasible. As a result, certificate-based attestation was used as a practical workaround. Despite these constraints, both project goals were achieved through a combination of adaptability, persistence, and focused problem-solving.

Future work could focus on enabling full hardware-based attestation using SEV-SNP reports with Azure's vTPM, completing the integration of a fully hardware-backed trust chain for remote attestation on Azure infrastructure. This would involve adapting Keylime's components in a manner similar to the SVSM-vTPM implementation. The manual verification process of Azure's attestation report, as documented in Section 4.3.7, may serve as a practical reference for identifying how the verification process needs to be implemented in code.

Additionally, further experimental research on the vTPM implementations of other cloud providers, such as AWS and GCP, would provide a more complete and accurate view of current confidential computing capabilities. In this project, the analysis of these platforms was based entirely on extensive yet fragmented documentation gathered from a wide range of sources. While this approach offered useful insights, hands-on experimentation could reveal important practical details that are not documented publicly.

As this field is evolving rapidly, it is also recommended to continuously monitor changes in cloud offerings, particularly around VM privilege levels, open-source support, and attestation mechanisms, to reassess the feasibility of designs like the SVSM-vTPM and refine independent attestation approaches accordingly.

Bibliography

[1] G. Cloud. (n.d.) What is cloud computing? [Accessed: 16-Mar-2025]. [Online]. Available: https://cloud.google.com/learn/what-is-cloud-computing

[2] E. Sayegh, "How cloud computing revolutionized business operations and what lies ahead," *Forbes*, 2023. [Online]. Available: https://www.forbes.com/sites/emilsayegh/2023/11/28/how-cloud-computing-revolutionized-business-operations-and-what-lies-ahead/

[3] A. Holmes, "Stolen data of 533 million facebook users leaked online," *Business Insider*, 2021. [Online]. Available: https://www.businessinsider.com/stolen-data-of-533-million-facebook-users-leaked-online-2021-4

[4] European Parliament and Council of the European Union, "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 (general data protection regulation)," Official Journal of the European Union, L 119, May 4, 2016, 2016. [Online]. Available: https://eur-lex.europa.eu/eli/reg/2016/679/oj

[5] U.S. Department of Health and Human Services, "Hipaa privacy rule to support reproductive health care privacy," Federal Register, Vol. 89, No. 81, April 26, 2024, 2024. [Online]. Available: https://www.federalregister.gov/documents/2024/04/26/2024-08503/hipaa-privacy-rule-to-support-reproductive-health-care-privacy

[6] MarketsandMarkets, "Confidential computing market by component (hardware, software, services), application (data security, secure enclaves, pellucidity between users), deployment mode, vertical (retail & consumer goods, bfsi) and region - global forecast to 2028," 2023. [Online]. Available: https://www.marketsandmarkets.com/Market-Reports/confidential-computing-market-27796261.html

[7] V. Narayanan, C. Carvalho, A. Ruocco, G. Almasi, J. Bottomley, M. Ye, T. Feldman-Fitzthum, D. Buono, H. Franke, and A. Burtsev, "Remote

attestation of confidential vms using ephemeral vtpms," in *Proceedings of the 39th Annual Computer Security Applications Conference*, ser. ACSAC '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 732–743. [Online]. Available: https://doi.org/10.1145/3627106.3627112

[8] R. Praveenraj and K. Desai, *Introduction to Confidential Computing: A Comprehensive Guide to Understanding and Exploring Confidential Computing*. New Delhi, India: BPB Publications, 2024.

[9] M. Peters and L. Sturmann, "Zero trust security with a hardware root of trust," Dec. 2023, accessed: 2025-03-16. [Online]. Available: https://www.redhat.com/en/blog/zero-trust-security-hardware-root

[10] A. Galanou, K. Bindlish, L. Preibsch, Y.-A. Pignolet, C. Fetzer, and R. Kapitza, "Trustworthy confidential virtual machines for the masses," 2024. [Online]. Available: https://arxiv.org/abs/2402.15277

[11] *Keylime Documentation*, [Accessed: Mar 5, 2025]. [Online]. Available: https://keylime.readthedocs.io/en/latest/

[12] "Svsm-vtpm artifacts," 2023, accessed: 2025-03-20. [Online]. Available: https://github.com/svsm-vtpm/SVSM-vTPM-artifacts

[13] J. Thijsman, M. Sebrechts, F. D. Turck, and B. Volckaert, "Trusting the cloud-native edge: Remotely attested kubernetes workers," 2024. [Online]. Available: https://arxiv.org/abs/2405.10131

[14] P. Mell and T. Grance, "The nist definition of cloud computing," National Institute of Standards and Technology, Tech. Rep. NIST Special Publication 800-145, 2011. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

[15] "What is a virtual machine and how does it work," Microsoft Azure, 2023, accessed: 2025-03-18. [Online]. Available: https://azure.microsoft.com/en-au/resources/cloud-computing-dictionary/what-is-a-virtual-machine/

[16] Y. Sun, D. Safford, M. Zohar, D. Pendarakis, Z. Gu, and T. Jaeger, "Security namespace: Making linux security frameworks available to containers," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1423–1439. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/sun

[17] W. Luo, Q. Shen, Y. Xia, and Z. Wu, "Container-IMA: A privacy-preserving integrity measurement architecture for containers," in *22nd*

International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), Beijing, China, 2019, pp. 487–500. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/luo

[18] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, vol. 1, 2015, pp. 57–64.

[19] O. Hosam and F. BinYuan, "A comprehensive analysis of trusted execution environments," in *2022 8th International Conference on Information Technology Trends (ITT)*, 2022, pp. 61–66.

[20] J. Nakajima, "Introduction to sgx (software guard extensions) and sgx virtualization," YouTube video, 2017, [Accessed: March 17, 2025]. [Online]. Available: https://www.youtube.com/watch?v=oES6hRhLJRM

[21] S. Mofrad, F. Zhang, S. Lu, and W. Shi, "A comparison study of intel sgx and amd memory encryption technology," in *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3214292.3214301

[22] D. Kaplan, J. Powell, and T. Woller, "AMD Memory Encryption White Paper," October 2021, [Accessed: March 17, 2025]. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf

[23] Advanced Micro Devices, Inc., "AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More," January 2020, [Accessed: March 17, 2025]. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf

[24] "Secure vm service module for sev-snp guests," Advanced Micro Devices, Tech. Rep. 58019 Rev. 0.50, 2023. [Online]. Available: https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/58019.pdf

[25] "linux-svsm: Linux SVSM (Secure VM Service Module)," https://github.com/AMDESE/linux-svsm, 2025, accessed: 2025-05-05.

[26] Microsoft, "Tpm fundamentals," 2024, [Accessed: March 19, 2025]. [Online]. Available: https://learn.microsoft.com/en-us/windows/security/hardware-security/tpm/tpm-fundamentals

[27] S. Berger, R. Cáceres, K. Goldman, R. Perez, R. Sailer, and L. van Doorn, "Vtpm: Virtualizing the trusted platform module," 07 2006.

[28] M. Bursell, "What measured boot and trusted boot means for linux," *Opensource.com*, 2020. [Online]. Available: https://opensource.com/article/20/10/measured-trusted-boot

[29] Sekyourity. (2024, Dec. 29) Secure boot, measured boot, and roots of trust. Accessed: 2025-05-27. [Online]. Available: https://medium.com/@sekyourityblog/do-you-know-who-your-computer-is-and-can-you-trust-it-4646b67deaf4

[30] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *13th USENIX Security Symposium (USENIX Security 04)*. San Diego, CA: USENIX Association, Aug. 2004. [Online]. Available: https://www.usenix.org/conference/13th-usenix-security-symposium/design-and-implementation-tcg-based-integrity-measurement

[31] N. Schear, P. T. Cable, T. M. Moyer, B. Richard, and R. Rudd, "Bootstrapping and maintaining trust in the cloud," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ser. ACSAC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 65–77. [Online]. Available: https://doi.org/10.1145/2991079.2991104

[32] "Keylime," [Accessed: Mar 5, 2025]. [Online]. Available: https://github.com/keylime

[33] M. Peters and G. Almasi, "Ibm implements remote attestation on linux with a hardware root-of-trust using keylime," July 2021. [Online]. Available: https://www.cncf.io/blog/2021/07/06/ibm-implements-remote-attestation-on-linux-with-a-hardware-root-of-trust-using-keylime/

[34] D. Duplyakin, R. Ricci, L. Stoller, M. Hibler, J. Duerig, K. W. Johnson, A. Akella, S. Alfeld, D. Arnold, and et al., "The design and operation of cloudlab," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019. [Online]. Available: https://www.usenix.org/system/files/atc19-duplyakin_0.pdf

[35] D. Buono and C. Carvalho, "Svsm-based vtpm implementation poc," 2022, accessed: 2025-05-05. [Online]. Available: https://github.com/svsm-vtpm/linux-svsm/blob/svsm-vtpm-preview/README-vtpm.md

[36] B. Posey. (2024) Top public cloud providers of 2025: How they compare. Accessed: 2025-05-05. [Online]. Available: https://www.techtarget.com/searchcloudcomputing/tip/Top-public-cloud-providers-A-brief-comparison

[37] "Amd sev-snp for amazon ec2 instances," Amazon Web Services, 2025, accessed: 2025-05-07. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/sev-snp.html

[38] "Enable amd sev-snp for an ec2 instance," Amazon Web Services, 2023. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/snp-work-launch.html

[39] "Uefi firmware github repository," Amazon Web Services, 2025, accessed: 2025-05-07. [Online]. Available: https://github.com/aws/uefi

[40] "The security design of the aws nitro system," Amazon Web Services, accessed: 2025-05-07. [Online]. Available: https://docs.aws.amazon.com/pdfs/whitepapers/latest/security-design-of-aws-nitro-system/security-design-of-aws-nitro-system.pdf

[41] "Amazon ec2 instances," Amazon Web Services, 2025, accessed: 2025-05-07. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/Instances.html

[42] "Azure confidential vm options," Microsoft Corporation, 2024, accessed: 2025-05-07. [Online]. Available: https://learn.microsoft.com/en-us/azure/confidential-computing/virtual-machine-options

[43] "Virtual tpms in azure confidential vms," Microsoft Corporation, 2023, accessed: 2025-05-07. [Online]. Available: https://learn.microsoft.com/en-us/azure/confidential-computing/virtual-tpms-in-azure-confidential-vm

[44] C. Perezvargas, "Confidential vms on azure," Microsoft Corporation, 2023, accessed: 2025-05-07. [Online]. Available: https://techcommunity.microsoft.com/blog/windowsosplatform/confidential-vms-on-azure/3836282

[45] "Azure confidential vms attestation guidance faq," Microsoft Corporation, 2024, accessed: 2025-05-07. [Online]. Available: https://github.com/Azure/confidential-computing-cvm-guest-attestation/blob/main/cvm-guest-attestation.md

[46] "Secure boot uefi keys - azure virtual machines," Microsoft, 2025, accessed: 2025-05-28. [Online]. Available: https://learn.microsoft.com/en-us/azure/virtual-machines/trusted-launch-secure-boot-custom-uefi

[47] "Hypervisor security on the azure fleet," Microsoft, 2022, accessed: 2025-05-28. [Online]. Available: https://learn.microsoft.com/en-us/azure/security/fundamentals/hypervisor

[48] "How to create a custom image for azure confidential vms," Microsoft, 2023, accessed: 2025-05-28. [Online]. Available: https://learn.microsoft.com/en-us/azure/confidential-computing/how-to-create-custom-image-confidential-vm

[49] "Supported configurations — confidential vm," Google Cloud, 2025, accessed: 2025-05-28. [Online]. Available: https://cloud.google.com/confidential-computing/confidential-vm/docs/supported-configurations

[50] "Oh SNP! VMs get even more confidential," Google Cloud, March 2023, accessed: 2025-05-28. [Online]. Available: https://cloud.google.com/blog/products/identity-security/rsa-snp-vm-more-confidential

[51] "New Confidential Computing updates for more hardware security options," Google Cloud, October 2024, accessed: 2025-05-28. [Online]. Available: https://cloud.google.com/blog/products/identity-security/new-confidential-computing-updates-for-more-hardware-security-options

[52] "Verify a confidential vm instance's firmware," Google Cloud, May 2025, accessed: 2025-05-28. [Online]. Available: https://cloud.google.com/confidential-computing/confidential-vm/docs/verify-firmware

[53] "Create custom confidential vm images," Google Cloud, May 2025, accessed: 2025-05-28. [Online]. Available: https://cloud.google.com/confidential-computing/confidential-vm/docs/create-custom-confidential-vm-images

[54] N. Thaker, "Deep dive into nitrotpm and uefi secure boot support in amazon ec2," December 2021, accessed: 2025-05-29. [Online]. Available: https://aws.amazon.com/blogs/compute/deep-dive-into-nitrotpm-and-uefi-secure-boot-support-in-amazon-ec2/

[55] "Confidential vm attestation," Google Cloud, 2025, accessed: 2025-05-29. [Online]. Available: https://cloud.google.com/confidential-computing/confidential-vm/docs/attestation

[56] "Retrieving endorsement keys," Google Cloud, 2025, accessed: 2025-05-29. [Online]. Available: https://cloud.google.com/compute/shielded-vm/docs/retrieving-endorsement-key

[57] salrashid123, "Sign, verify and decode using google cloud vtpm endorsement and attestation key and certificate," 2025, accessed: 2025-05-29. [Online]. Available: https://github.com/salrashid123/gcp-vtpm-ek-ak/blob/main/README.md

[58] AWS_Sam, "Answer to: How to provision nitrotpm," 2023, accessed: 2025-05-29. [Online]. Available: https://repost.aws/questions/QUWzHj71c4TTWDbYP5RYCwHA/how-to-provision-nitrotpm

[59] "Retrieve the public endorsement key for an ec2 instance," Amazon Web Services, 2025, accessed: 2025-05-07. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/retrieve-ekpub.html

[60] "Attest an amazon ec2 instance with amd sev-snp," Amazon Web Services, 2025, accessed: 2025-05-29. [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/snp-attestation.html

[61] "Azure confidential virtual machines faq," Microsoft, 2025, accessed: 2025-05-29. [Online]. Available: https://learn.microsoft.com/en-us/azure/confidential-computing/confidential-vm-faq

[62] C. Perezvargas, "Openhcl: the new, open source paravisor," 2024, accessed: 2025-05-29. [Online]. Available: https://techcommunity.microsoft.com/blog/windowsosplatform/openhcl-the-new-open-source-paravisor/4273172

[63] "Shielded vm," Google Cloud, 2025, accessed: 2025-05-29. [Online]. Available: https://cloud.google.com/compute/shielded-vm/docs/shielded-vm

[64] W.-K. E. Belemgnegre. (2024, Jun.) The ultimate guide to docker: Benefits, architecture, and practical steps. Accessed: 2025-05-29. [Online]. Available: https://medium.com/@belemgnegreetienne/the-ultimate-guide-to-docker-benefits-architecture-and-practical-steps-02d0f02e7eee

[65] B. Sandu. (2025, Apr.) What is docker compose? simplifying multi-container apps. Accessed: 2025-05-29. [Online]. Available: https://tms-outsource.com/blog/posts/what-is-docker-compose/?utm_source=chatgpt.com

[66] "Rust implementation of the keylime agent," https://github.com/keylime/rust-keylime, Keylime, 2025, accessed: 2025-05-29.

[67] W. Warley. (2024) Mastering vagrant: A practical guide to building and managing virtual development environments. Accessed: 2025-05-30. [Online]. Available: https://medium.com/@williamwarley/mastering-vagrant-a-practical-guide-to-building-and-managing-virtual-development-environments

[68] IBM, "Ima 1.0 documentation," 2023, accessed: 2025-05-31. [Online]. Available: https://ima-doc.readthedocs.io/en/latest/

[69] Keylime, "Enhancement 23: Use a dedicated keylime user account," 2020, accessed: 2025-05-31. [Online]. Available: https://github.com/keylime/enhancements/blob/master/23_dedicated_keylime_user_account.md

[70] Red Hat, Inc. (2024) Ensuring system integrity with keylime. Accessed: 2025-06-01. [Online]. Available: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/9/html/security_hardening/assembly_ensuring-system-integrity-with-keylime_security-hardening

[71] K. S. Pujari, "Terraform vs pulumi: I used both—here's what i'd choose again," 2022, accessed: 2025-06-02. [Online]. Available: https://medium.com/@kanishksinghpujari/terraform-vs-pulumi-i-used-both-heres-what-i-d-choose-again-542a5b2bcae7

[72] Pulumi, "Deploy a virtual machine to azure," 2025, accessed: 2025-06-02. [Online]. Available: https://www.pulumi.com/templates/virtual-machine/azure/

[73] "Azure confidential vm guest attestation design detail," Microsoft, February 2025, accessed: 2025-06-03. [Online]. Available: https://learn.microsoft.com/en-us/azure/confidential-computing/guest-attestation-confidential-virtual-machines-design

[74] Trusted Computing Group, "Registry of reserved tpm 2.0 handles and localities," 2017, accessed: 2025-06-03. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/RegistryOfReservedTPM2HandlesAndLocalities_v1p1_pub.pdf

[75] VPSBG, "How to perform amd sev-snp attestation inside a guest virtual machine," April 2025, accessed: 2025-06-03. [Online]. Available: https://www.vpsbg.eu/docs/how-to-perform-amd-sev-snp-attestation-inside-a-guest-virtual-machine

[76] "Faq for trusted launch - azure virtual machines," Microsoft, May 2025, accessed: 2025-06-03. [Online]. Available: https://learn.microsoft.com/en-us/azure/virtual-machines/trusted-launch-faq