Trust in Lightweight Virtual Machines: Integrating TPMs into Firecracker

Master's thesis in Computer science and engineering

Alexandra Parkegren, Melker Veltman

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Trust in Lightweight Virtual Machines: Integrating TPMs into Firecracker

Alexandra Parkegren, Melker Veltman

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Trust in Lightweight Virtual Machines: Integrating TPMs into Firecracker

ALEXANDRA PARKEGREN, Chalmers University of Technology
alepar@chalmers.se

MELKER VELTMAN, Chalmers University of Technology
melkerv@chalmers.se

© ALEXANDRA PARKEGREN, 2023.
© MELKER VELTMAN, 2023.

Supervisor: Victor Morel, Department of Computer Science and Engineering
Advisor: Erik Dahlgren, Scionova AB
Examiner: Ahmed Ali-Eldin Hassan, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Abstract

Due to the rise of service-based software products, cloud computing has seen significant growth in recent years. When software services use cloud providers to run their workloads, they place implicit trust in the cloud provider, without any explicit trust relationship. One way to achieve such explicit trust in a computer system is to use a hardware Trusted Platform Module (TPM), which is a coprocessor for secure cryptographic functionality. However, in the case of managed platform-as-a-service offerings, there is currently no provider exposing the trusted computing capabilities of a TPM.

The main goal of this project is to enable system designers to improve trust by providing access to a TPM within a cloud-based environment. This was achieved by integrating a TPM device into the Firecracker hypervisor, originally developed by Amazon Web Services. In addition to this, multiple performance tests along with an attack surface analysis were performed to evaluate the impact of the changes introduced.

The results show a significant performance impact; however, by using a resource pool, they could be partially mitigated. The analysis of the attack surface shows that there is no major change in the Firecracker hypervisor itself. However, the attack surface is extended by allowing cloud users to communicate with a TPM. Therefore, we discuss the impact and possible mitigations of the increased attack surface. Then we describe what it takes for a cloud service provider to offer trusted computing capabilities to its customers. Lastly, we conclude that the slight performance decrease along with the attack surface increase should be acceptable trade-offs in order to enable trusted computing in platform-as-a-service offerings.

Keywords: Trust, TPM, Virtualisation, Firecracker, Linux, Platform-as-a-Service, Cloud

Acknowledgements

We want to express our gratitude to our supervisor Victor Morel, whose guidance, feedback, and support have been invaluable throughout the duration of this master thesis. We also want to express our appreciation for all the support, help, and resources provided by Scionova AB. Especially we would like to thank our company supervisor Erik Dahlgren as well as Daniel Hemberg. Furthermore, we thank our examiner Ahmed Ali-Eldin Hassan for providing us with feedback throughout the project.

Alexandra Parkegren, Gothenburg, June 2023

Melker Veltman, Gothenburg, June 2023

Contents

List of Acronyms **xi**

List of Figures **xiii**

List of Tables **xv**

List of Listings **xvii**

1 Introduction **1**
- 1.1 Aim . . . 1
  - 1.1.1 Research Question . . . 2
  - 1.1.2 Scope . . . 3
- 1.2 Thesis Outline . . . 3

2 Background **5**
- 2.1 Trusted Computing . . . 5
  - 2.1.1 Trusted Platform Module . . . 5
  - 2.1.2 Root of Trust for Measurement . . . 6
  - 2.1.3 Integrity Measurement Architecture . . . 6
  - 2.1.4 Remote Attestation . . . 7
  - 2.1.5 Trusted Execution Environments . . . 8
- 2.2 Virtualisation . . . 9
  - 2.2.1 Types of Virtualisation . . . 9
  - 2.2.2 Lightweight Virtualisation . . . 10
- 2.3 Cloud Computing . . . 10
  - 2.3.1 Service Models . . . 11
  - 2.3.2 Achieving Trust in the Cloud . . . 12

3 Related Work **13**
- 3.1 VTPM Implementation Models . . . 13
- 3.2 Trust in Cloud Computing Environments . . . 14

4 Method **17**
- 4.1 Using TPMs in a Lightweight VMM . . . 17
- 4.2 Required VMM Changes . . . 18
- 4.3 Test Configuration . . . 18

4.4 Attack Surface Analysis . . . 20

5 Implementation **21**
5.1 Firecracker TPM Device . . . 21
5.2 Test Environment Setup . . . 22
5.3 Test System Design . . . 23
5.4 Test Suite Tools . . . 24
5.5 Pooling Algorithm . . . 25
5.6 Starting a Firecracker VM with a TPM . . . 26
5.6.1 Prerequisites . . . 26
5.6.2 Compiling Firecracker and Building a Linux Kernel . . . 27
5.6.3 Starting a vTPM and Running a VM . . . 27

6 Results **29**
6.1 Performance overhead . . . 29
6.2 Attack Surface Analysis . . . 32

7 Discussion **35**
7.1 Startup Time Evaluation . . . 35
7.1.1 Caveats in a Concurrent Environment . . . 35
7.1.2 Differences to Previous Work . . . 36
7.1.3 Resource Pool . . . 36
7.2 Memory Overhead Evaluation . . . 37
7.2.1 Relation to VM Memory Sizes . . . 37
7.2.2 Compared to Other VMMs . . . 37
7.3 Security Evaluation . . . 37
7.3.1 Isolating with Linux Kernel Functionality . . . 38
7.3.2 Alternative Isolation Methods . . . 38
7.4 Practical Application . . . 38
7.5 Future work . . . 39
7.6 Ethical Considerations . . . 40

8 Conclusion **41**

Bibliography **43**

A Extra Figures **I**
A.1 Memory Overhead Result Chart . . . I
A.2 Firecracker TPM Sequence Diagram . . . II

B Pooling Algorithm Implementation **III**

List of Acronyms

AWS Amazon Web Services
CDF Cumulative Distribution Function
CoCoTPM Confidential Computing TPM
crosvm Chrome OS Virtual Machine Monitor
CRTM Core Root of Trust for Measurement
CSP Cloud Service Providers
FaaS Function as a Service
IaaS Infrastructure as a Service
IBM International Business Machines Corporation
IMA Integrity Measurement Architecture
KVM Linux Kernel Virtual Machine
PaaS Platform as a Service
PCR Platform Configuration Register
QEMU Quick EMUlator
RTM Root of Trust for Measurement
SaaS Software as a Service
SEV Secure Encrypted Virtualization
SGX Software Guard Extensions
SvTPM Secure vTPM
syscalls System Calls
TCCP Trusted Cloud Computing Platform
TCG Trusted Computing Group
TEE Trusted Execution Environment
TPM Trusted Platform Module
VM Virtual Machine
VMM Virtual Machine Monitor
vTPM Virtual TPM

List of Figures

2.1 Startup Integrity Measurement Process 7
2.2 Remote Attestation Protocol 8
2.3 Linux Virtualisation Software Stack 11

4.1 Measurement System Interactions 18
4.2 Firecracker Device Model with TPM Support 19

5.1 Class Diagram of Firecracker Changes 21
5.2 Test System Architecture Diagram 23

6.1 Distribution of Boot Times for 500 VMs 30
6.2 Distribution of Boot Times for 1000 VMs 31

List of Tables

6.1 Boot Times for 500 VMs 30
6.2 Boot Times for 1000 VMs 31
6.3 Memory Overhead Results 32
6.4 Changes in Lines of Code 32

List of Listings

1 TPM Allocator Interface . . . 23
2 Performance Test Help Text . . . 24
3 *Pmap* Example Output . . . 25
4 Resource Allocator Interface . . . 26

1

Introduction

Cloud computing has transformed the way we store, access, and process data. The ability to access and use computing resources on demand, without the need for large upfront investments in hardware and infrastructure, has made cloud computing an attractive and flexible option for businesses of all sizes [1]. Another purpose of using cloud technologies is the ability to hand over some management to the cloud provider, reducing the user's responsibility. Pichan *et al.* [2] argue for the relevance and importance of cloud computing, how it has changed the way Internet services are consumed, its economic opportunities, and how cloud-based solutions are expected to see greater adoption in the future.

By transferring on-site infrastructure to cloud computing platforms, more control and trust is given to the third-party company hosting the service. In practice, companies let cloud providers execute their software and process potentially sensitive data of their customers. To secure both the system running and the handling of data, safe storage and verification of the platform should be implemented to ensure trust properties. The potential adversary in this case is both the cloud provider itself and other customers of the cloud provider who execute code on the same physical hardware.

One way to achieve trust in a computer system is to use a *Trusted Platform Module* (TPM), which is a coprocessor for secure cryptographic support. The TPM interfaces with the CPU through a specific set of instructions and can authenticate the environment in which the application is executed.

In the case of on-site infrastructure, a TPM is implemented in hardware. Considering cloud infrastructure, the customer is most often given a *virtual machine* (VM), which can be equipped with a software emulation of a TPM. This software TPM is also considered trusted, as there are multiple ways to associate such a TPM with trusted hardware [3]. However, in the case where cloud providers maintain both the machine as well as the operating system, there is no one who exposes *trusted computing* capabilities such as a software TPM.

1.1 Aim

The main goal of the project is to incorporate TPM capabilities for an isolated and performant virtualised computing environment while maintaining a chain of trust,

the safe properties of a TPM, and the performance properties of the computing environment. Virtual machines driven by the Firecracker [4] *Virtual Machine Monitor* (VMM) will be used as the computing environment for the project. Firecracker is used in offerings by *Amazon Web Services* (AWS), where they maintain the underlying server and infrastructure. Firecracker was created with the goals of workload isolation, low startup times, and low memory overhead. To achieve minimal overhead, several trade offs have been made in comparison to traditional VMMs. One of the delimitations is not to support as many emulated devices for the VMs, as this support is considered an unnecessary overhead for the use case of Firecracker. Because of the minimal support for devices, there is no current support for emulating a TPM in Firecracker.

Incorporating a TPM in such an environment might have an impact on said performance properties such as memory overhead and startup times. However, to what degree is unknown and will be examined in this project. The nature of the workloads also means that Firecracker has to be able to handle sudden bursts of computation, such as multiple VMs starting up at the same time. Since one of the goals of Firecracker is to be able to dynamically change workloads [4], software TPMs must be allocated to VMs as they are created. This allocation poses a potential performance decrease in the startup time. However, we do see a potential to mitigate this performace decrease by using a resource pool with software TPMs, reducing the need to instantiate new ones as workloads changes. Regarding the security goals of Firecracker, the attack surface of the VM will also change as a software component executed on the host interfaces with a VM.

The basis of this project is therefore also to enable system designers to further enhance trust by giving them access to a TPM within cloud-based environments. By describing the benefits of TPMs within cloud workloads and displaying an example of their use in Firecracker, we encourage system designers to verify the integrity of their cloud environment. By using and contributing to the open source community, and continuing the sharing of information and resources to both users and cloud providers, the work can be easy to access for anyone to evaluate and develop further.

1.1.1 Research Question

The high-level question is formulated as follows:

> What is the impact on performance and security when incorporating software TPMs in lightweight VMs and how can it be mitigated?

To further concretise this question, it was split into the following research questions:

1. How does the startup time and memory overhead of a lightweight VM change if a software TPM is created and allocated to it?
2. What impact on the startup time does maintaining a resource pool of TPMs to be allocated have?
3. How does incorporating a TPM change the attack surface of a lightweight VM?

Significant preparations are required for both questions (1) and (2). The first challenge is to create and incorporate TPMs into Firecracker. Then a measurement system needs to be designed and implemented to record startup times and memory overhead.

The main challenges reside in the first part. As lightweight virtual machines use a minimal feature set, an implementation of software TPMs for Firecracker will be different from conventional virtual machines.

Regarding the measurement system, several scenarios have to be created to simulate different use cases and workloads. These scenarios will be based on the performance tests used in the published research article on Firecracker [4]. The measurement system will be designed from the standpoint that the tests should be reproducible, to encourage further research within the field.

Regarding question (3), the challenge is to examine the current attack surface of the lightweight VM and compare it to a lightweight VM with a TPM device. Although the capabilities of a lightweight VM are limited, it still has a reasonably large device interfacing surface, which has to be taken into account.

1.1.2 Scope

As the main contribution of the project is the incorporation of TPMs into Firecracker, the surrounding and supporting services are only considered examples. These services include the simulated manufacturing of TPMs, the TPM pooling service, and the VM instantiation service. In other words, a complete workload manager is a component that a cloud provider would create internally to better fit our contribution into their software ecosystem.

As this project aims to enable the use of TPMs for application developers, it focuses on the platform part of a cloud-based service that uses Firecracker. The application of a TPM is beyond the scope of this project, as it is an already researched topic [5].

1.2 Thesis Outline

The thesis is structured as follows. This chapter serves as an introduction to the subject, providing relevant background information and the motivation behind the chosen thesis topic. It also presents the research questions that will be addressed throughout the thesis. Chapter 2 introduces important concepts that are necessary to understand the various building blocks required for the new implementation. In chapter 3, an overview of the current research landscape is presented, highlighting the existing work that has been conducted in the field.

Chapter 4 explains the methods that will be employed to address the research questions. Chapter 5 then explains the implementation in more detail, providing a comprehensive explanation of how the different components interact and can be used. Chapter 6 presents the measured results which are further discussed in chapter 7,

where possible improvements and potential future work are also explored. Finally, chapter 8 concludes the thesis by summarising key findings.

2

Background

This chapter describes and explains technical concepts related to trusted computing, virtualisation, and cloud computing. The relation between these topics is also explained to put emphasis on the industry relevance of the project.

2.1 Trusted Computing

The expression *trusted computing* has been apparent in research since the early 2000s and received organisational support from the Trusted Computing Platform Alliance, which later became the *Trusted Computing Group* (TCG). The TCG is the current governing entity for specifications and standards regarding trusted computing.

Trusted computing can be compared to computer security in the sense that it relates to whether a system behaves as expected. However, the difference lies in whether the user implicitly trusts the hardware. Computer security describes the security, given that the hardware can be trusted, whereas trusted computing describes *how* the hardware can be trusted. When modern cloud computing is taken into account, there is a growing overlap between software and hardware, as well as between the concepts of trusted computing and computer security [6, pp. 2-3].

2.1.1 Trusted Platform Module

One of the main commitments of the TCG is the maintenance and development of the specifications and standards for the TPM. The TPM is a hardware module that provides functionality for cryptography, along with random number generation, cryptographic hashes, and secure storage for small amounts of data [6, pp. 35-37]. TPMs are present in all modern computers, and as of Microsoft Windows 11, a TPM compliant with the TPM 2.0 specification is a system requirement [7].

The manufacturing of a TPM must be carried out by a certified manufacturer according to the TCG specification. When a TPM is manufactured, the manufacturer supplies it with keys and certificates, which proves its authenticity and establishes a root of trust. Then the TPM manufacturer hands the TPM chip to a platform manufacturer, who could be a laptop manufacturer, as an example. The platform manufacturer provisions more certificates in the TPM, proving the authenticity of the platform with that specific TPM. The result of this process is that the end user can verify that the platform was manufactured with a TPM, which, in turn, was

manufactured by a certified manufacturer. Combining this verification and the fact that the TPM is a physically separate processor with a limited instruction set enables the user to trust the computation made in the TPM [8, pp. 305-307].

Included in the secure storage functionality of a TPM is the *Platform Configuration Register* (PCR). PCRs are shielded locations in the TPM used to store data adhering to specific write semantics. When a user wants to write to a PCR, it can only be done using a specific *extend* operation. This operation produces a *digest* using a secure hashing algorithm, with the new value and the current value of the register. This digest is then stored in the PCR [9, pp. 184-185]. A common use case of a PCR is to *measure* the code running in a system. To measure in this scenario means computing a digest, also called a measurement, of the software component and extending the PCR with that value [8, pp. 156-157].

TPMs can also be implemented in software; however, as the manufacturing now occurs at the start of a software program, a trust establishment procedure must be completed. This establishment is a challenge that has been researched before, where some solutions use the hardware TPM of the host to produce the keys needed for the software TPM. In effect, the software TPM may achieve a similar root of trust as the hardware TPM, but software-implemented TPMs are considered less trusted compared to hardware TPMs. A TPM implemented in software can also be called *virtual TPM* (vTPM), where a common implementation is the *swtpm* made by *International Business Machines Corporation* (IBM) [3]. As the trust establishment procedure uses a hardware TPM, it may pose a performance limit if the system has a high demand for vTPMs [3].

2.1.2 Root of Trust for Measurement

To be able to measure software into the PCR, a *Root of Trust for Measurement* (RTM) must be established. The RTM is a component that measures the first software executed in a system. With PCs in mind, the RTM is the CPU, as it executes the first pieces of code. The code executed is the *Core Root of Trust for Measurement* (CRTM), which is stored in read only memory within the platform. Both the CRTM and the RTM are considered implicitly trusted [10, p. 53].

When the RTM has measured the first piece of software, it can make a decision whether to execute it or not. This decision is based on a policy setting and the measurement itself, as it can be set up to only execute software producing a specific hash digest. Given that the software starts executing, control is handed over to it, and that software is now responsible for measuring the next component. This process builds a chain of trust that originates at the RTM. An example of the measurement process of a system can be seen in Figure 2.1.

2.1.3 Integrity Measurement Architecture

One implementation of a system capable of producing such a chain of trust is the Linux kernel with its *Integrity Measurement Architecture* (IMA). The IMA can be configured with policies to declare which parts of the system should be measured.

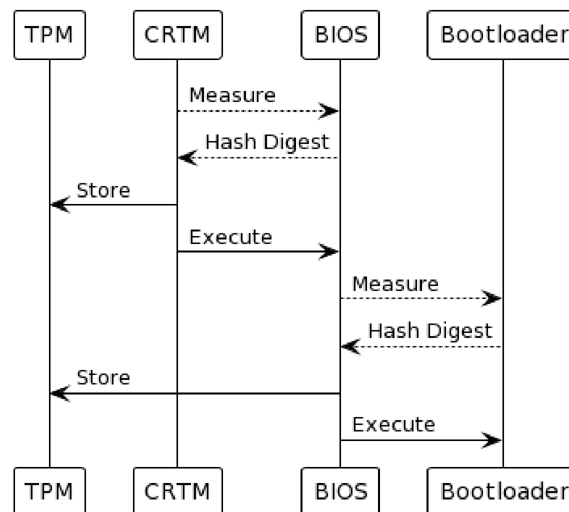


Figure 2.1: Early startup integrity measurement process

Using such policies, the IMA can be set up to measure and verify all kernel modules loaded as an example. Another example is to measure all files executed by the root user.

When an object is measured by the IMA, the PCR is extended with the hash digest of the object and it is added to a measurement list. Since the PCR value is a combined hash of all the values in the measurement list, the list can be trusted if the TPM is trusted and the PCR value matches [5].

2.1.4 Remote Attestation

Attestation is a security mechanism that enables an end user to verify the integrity of the system. When the user does not reside on the same platform, the mechanism is also remote. Combining attestation with integrity measurements enables a user to place trust in a system without physical access to it.

The TCG provides an integrity reporting protocol as part of their Specification Architecture Overview [11, pp. 5, 9–10]. As this integrity reporting protocol is general, some components must be added for security reasons. The process includes two parties: an attester who wants to attest a system and the system to be verified. A simple example of an attesting system and the attester is a mail service provider and an end user of such a service. It works as follows:

1. The verifier generates a new, non-predictable, non-repeating, random value, called *nonce*.
2. The verifier sends a challenge request including the nonce to the attestation service in the attestation system.
3. The attestation service asks its TPM to sign and return the PCR values along with the nonce received, which is called a PCR quote.
4. The TPM in the attesting system concatenates the received nonce with the

PCR value, signs it using a key and then returns the signature and the signed data.

5. The attestation service requests the measurements from the measurement list.
6. The measurement list returns the measurements.
7. The attestation service sends a challenge response containing the TPM quote and the measurement list.
8. As the PCR value can be calculated based on the measurement list and the signature inside the quote verifies the PCR, the verifier can now validate that the reported measurement list is trusted.
9. The last step is for the attester to check if the measurement list corresponds to the previously stored measurements.

The nonce is often only valid for a certain amount of time, which ensures transmittal of live data and protects from replay attacks. This attestation process [5] can be seen in Figure 2.2.

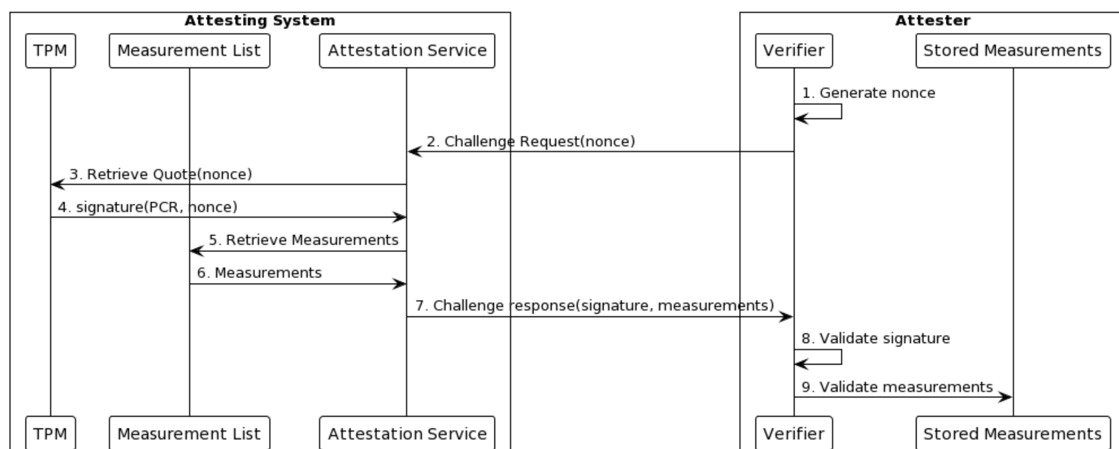


Figure 2.2: Remote Attestation Protocol

2.1.5 Trusted Execution Environments

A disadvantage of the TPM is that it cannot support arbitrary programs. It only supports a predefined set of instructions and actions to perform. In response to this, a *Trusted Execution Environment* (TEE) offers a secure and isolated execution environment for arbitrary programs. A TEE puts several guarantees on its execution and runtime in comparison to a general purpose environment. These guarantees include authenticity of executed code, integrity of runtime states among others [12].

TEEs are not a standard and therefore implementations differ significantly between vendors. Intel offers a TEEs called *Software Guard Extensions* (SGX), that can be used to instantiate and execute an *enclave*. An enclave is an instance of a TEE, in the sense that it is a set of resources that consists of code and data, executed in the same domain under shared protection. Therefore, using an enclave entails executing

sensitive purpose-built programs isolated from the rest of the system. Another approach is AMDs *Secure Encrypted Virtualization* (SEV), which enables the use of encrypted virtual machines supported by hardware. SEV acts as an isolation barrier between the host OS and the virtual machine, which means that the host cannot examine the memory of the virtual machine [13].

2.2 Virtualisation

The concept of virtualisation dates back to long before cloud computing came about and laid the foundation for the computing needs of today. It started out as a solution to the time sharing problem of mainframe computers manufactured by the IBM [14]. Further on, VMWare filed a patent for a 'Virtualization system including a virtual machine monitor for a computer with a segmented architecture' [15], resulting in them taking a leading role in the virtualisation industry. However, several other options have been available despite the patent. In essence, virtualisation creates an abstraction of the hardware to achieve some goal, where which goal to achieve depends on the virtualisation technology. The main goals of virtualisation can be seen below.

Improve resource utilisation Run multiple workloads on the same physical hardware

Workload isolation Mitigate interference between different workloads

Reduce hardware needs By virtualising specific hardware for developers, the need for dedicated computing devices decreases

Compatibility Ensure a consistent deployment model with virtualisation

Different virtualisation solutions target different goals.

The software component responsible for creating and running virtualised environments is called *hypervisor* or VMM. The virtualised environment is usually called a virtual machine, comparable to the concept of a physical machine [16].

2.2.1 Types of Virtualisation

Executing software in an abstracted environment through virtualisation can be achieved at different levels. At one end, there is full virtualisation or emulation. This method runs the exact same software as would be in a physical environment, and every emulated device is exposed as a hardware device from inside the VM.

Another method is to use paravirtualisation, popularised by the Xen VMM [17]. This method exposes a different set of hardware, which needs specific drivers to be used by the operating system. However, what is lost in compatibility is gained in performance, since paravirtualised hardware performs better than their emulated counterpart [18, p. 43-46]. Paravirtualised hardware has been standardised with the *virtio* project [19], which also adds Linux kernel primitives to support future Linux driver development and VMM support. Another hypervisor is *Quick EMUlator*

(QEMU), which can use both Xen and the Linux *Kernel Virtual Machine* (KVM) as backends [20].

On the other side of the virtualisation space is *OS-level virtualisation* or *containerisation*. The main difference from other types of virtualisation is that OS-level virtualisation does not run a complete operating system for each workload. Instead, it uses the underlying OS for all workloads on one machine. As each workload uses the same underlying kernel, isolation is decreased, which also means a decrease in security with regard to workload interference. OS-level virtualisation commonly uses kernel functions such as namespaces and cgroups for isolation [21]. Linux namespaces can expose an isolated abstraction of resources to a set of processes. An example is a network namespace that isolates the network stack and the ports of the set of processes using the namespace [22]. Linux cgroups is short for control groups and can be used to limit memory usage and CPU usage for a set of processes [23].

One widely adopted containerisation solution is Docker [24], which is a composition of tools to support containers, build environments, and a consistent deployment model across various physical hardware. Another similar solution is Podman [25].

2.2.2 Lightweight Virtualisation

In recent years, the virtualisation space has seen development from major cloud actors such as Google and Amazon, with the purpose of minimising resource overhead and increasing workload isolation. The project made by Google is called *gVisor*, and builds upon container technology to increase workload isolation through system call overloading. *Firecracker* is the name of the project created by AWS and is used in serverless environments. The developer interface of the two solutions is similar, but the technologies supporting them are different.

Firecracker uses KVM and therefore tries to reduce resource overhead while maintaining the isolation properties of paravirtualisation [26]. The main difference between Firecracker and conventional hypervisors is the amount of devices it supports. By decreasing the complexity, Firecracker is able to improve boot times and decrease memory overhead [4]. Firecracker is implemented in Rust and uses a set of virtualisation primitives built by Intel, Google, and Amazon. These virtualisation primitives are distributed as Rust libraries and are used in virtualisation projects at all three corporations, but with different goals. The implementation at Intel is called *Cloud Hypervisor* and the one at Google is called *Chrome OS Virtual Machine Monitor* (crosvm). The virtualisation primitives make use of KVM in a way similar to QEMU [4, 27, 28]. An overview of the relationship between the different projects and software components can be seen in Figure 2.3.

2.3 Cloud Computing

In essence, cloud computing can be considered as the action of buying the usage of a resource used within the computing services owned by someone else. Considering the agile and fast-paced environments of tech businesses today, the need for instant

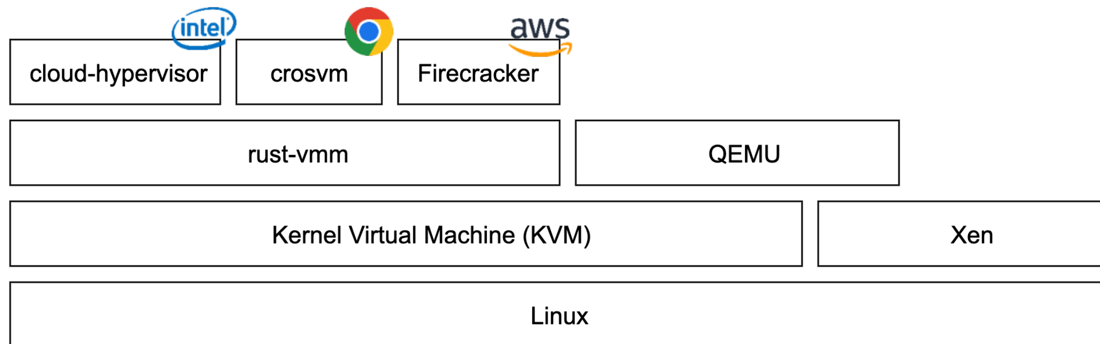


Figure 2.3: Linux virtualisation software stack

computation power to test and verify new ideas is greater than ever. By letting other companies be responsible for hardware, a purchase process can be shortened from months to hours, enabling the success of such tech businesses [29, p. 24-41].

2.3.1 Service Models

Architecting cloud solutions today involves many choices based on the application and the environment within which it should be maintained. Previously, cloud computing solely consisted of the ability to rent computing power, whereas today, *Cloud Service Providers* (CSP) provide a wider portfolio of services. As an example, AWS as of 2023 offers more than 200 different services [30]. These services can be divided into three main service models:

- *Infrastructure as a Service* (IaaS)
- *Platform as a Service* (PaaS)
- *Software as a Service* (SaaS)

IaaS offerings give the user access to computer infrastructure such as computing units, storage, or networking. The infrastructure still has to be configured like physical machines, however the configuration is done through software instead of hardware. The maintenance model of an IaaS solution gives the user a lot of freedom and responsibility to use customised network topologies or custom operating systems as examples. Examples of IaaS offerings in AWS are *Elastic Compute Cloud* for computing, *Elastic Block Storage* for storage, and *Virtual Private Cloud* for networking.

In contrast to IaaS, PaaS abstracts the system up to the application layer. Therefore, the user of this model does not have access to raw storage, network configuration, or the operating system. With the PaaS model, a developer only has to deliver a software package along with some deployment configuration such as memory size to deploy an application. When using PaaS offerings, cross-cutting concerns, such as

metrics, error reporting, and alerts, can be automatically set up by the CSP, reducing the maintenance need for users. To account for dynamic load scaling, the workload in the eye of the CSP is assumed to be ephemeral. The *Function as a Service* (FaaS) model also falls into this space, where a function is delivered to the CSP and the platform manages invocation based on some event. In FaaS, the ephemeral property is taken to the edge where each execution supports only one invocation [4]. AWS offers multiple PaaS service for different use cases, these include Fargate and Lambda.

On the other side in comparison to IaaS is SaaS, which is hosted services targeting end users. Here, the user has limited configuration ability and even less responsibility with regards to maintenance. Services in this space include Microsoft Office 365 and Google Gmail [29, p. 44-59].

2.3.2 Achieving Trust in the Cloud

When considering trust in relation to cloud computing service models, considering the major CSPs, there is currently only support in the IaaS model. However, AWS provide an option to attach a *NitroTPM* to some of their virtual machines. NitroTPM is a TPM implementation that complies with the TPM 2.0 specification. But they have not published information regarding whether the component is implemented in hardware or software [31].

In *Google Cloud Platform*, they offer *Shielded VMs* which include virtual TPMs and therefore enable the use of integrity measurements among other functionalities [32]. *Microsoft Azure*, offers a similar service called *Trusted Launch for Virtual Machines* [33]. What is apparent from considering these products is that TPMs are widely available in IaaS models, whereas similar functionality does not exist in the other models.

3

Related Work

This chapter is dedicated to explaining the current state-of-the-art in the space of trusted cloud computing. It includes research regarding specific systems for attesting integrity in cloud environments and a discussion on recent software implementations of TPMs. It also includes the integration of trusted computing primitives, such as TPMs, in cloud environments.

3.1 VTPM Implementation Models

To improve performance, avoid data leakage and isolation issues in using a vTPM, Wang *et al.* [34] continued the work of Berger *et al.* [3] and created an alternative TPM software implementation, called *Secure vTPM* (SvTPM), not to be confused with swtpm. The novelty of their work is to base the trust in a TEE instead of a hardware TPM. More specifically, they use Intel SGX to run or protect specific parts of the TPM in an SGX enclave. The result of their work is a solution where the startup time is decreased by multiple orders of magnitude in comparison to previous implementations. However, since Intel SGX is a proprietary Intel technology, it may not be applicable to cloud computing environments. With this in mind, initiatives such as HyperEnclave [35] can be used to make such a solution cross-platform. HyperEnclave uses the integrity measurement and attestation capabilities of a TPM to extend the chain of trust to the TEE.

Another approach, called *Confidential Computing TPM* (CoCoTPM), uses AMD SEV to achieve similar goals. The goal for CoCoTPM was to minimise the trust required towards the host and the hypervisor, in a virtualised environment. In the article, Pecholt and Wessel [36] apply their methods in cloud environments to support confidential computing needs. The main idea is to run vTPMs in an encrypted VM, using AMD SEV, and connect these TPMs to encrypted VMs running the desired workloads. Communication between workload and TPM is encrypted and integrity protected, further increasing the confidentiality property of their work [36].

Contrary to the goals of a vTPM, security issues in the implementation could be an entry way for adversaries. A recent vulnerability was discovered in which the lack of a proper length check and proper size handling could lead to an out-of-bounds memory read and write. The out-of-bounds memory write vulnerability could also lead to arbitrary code execution [37, 38, 39]. Taking the vulnerability into account with regards to vTPM, SvTPM and CoCoTPM highlights the need of applying

defence-in-depth principles, as an adversary potentially could execute code within the environment that the vTPM is executing. In the case of using a vTPM, the environment would be the host, in the case of using a SvTPM, the answer is not clear and would need to be researched further. Considering the use case for a CoCoTPM, the environment would be the encrypted VM where the TPMs of the other workloads on the same host would reside, which can be considered quite critical. However, as these TPMs are implemented in software, the process of applying security patches is shorter compared to hardware TPMs.

3.2 Trust in Cloud Computing Environments

In the literature study on trust in cloud computing, Ibrahim and Hemayed [40] highlight the significant role of the TPM. They also investigate different aspects of various architectures and managers for IaaS services. They further compare different implementations and types of virtual TPMs and remote attestation types and discuss secure boot and integrity monitoring, all applied to cloud computing.

An interesting discussion and proposal are made by Santos *et al.* [41], where they describe a design of a *trusted cloud computing platform* (TCCP). The work discusses a system for trusted cloud computing through a distributed approach that matches the requirements and resources available at CSPs. They use remote attestation in a distributed network and stand on previously researched primitives regarding confidential computing to hide the data of a VM to the host OS [42].

If we now consider PaaS offerings, CSPs are required to make a decision whether to use virtualisation or containerisation, where virtualisation in this sense does not include OS-level virtualisation. Both options come with shortcomings and strengths, with virtualisation being more heavyweight but well isolated, and containerisation more lightweight but less isolated. This comparison has seen a lot of research, both on performance and isolation [43, 44, 45, 46].

Recent development within the space of runtimes for PaaS offerings has been seen with Google gVisor, from the container side, and AWS Firecracker, from the virtualisation side. Young *et al.* [47] made a study comparing the gVisor runtime, runsc with the default Docker runtime, runc. Their results are noteworthy, as they show a significant decrease in performance with respect to system calls, memory allocations, and networking. Another comparison between Firecracker and gVisor has been made by Anjali *et al.* [26], arriving at a result in favour of Firecracker for network bandwidth, memory allocation times, and file access.

Firecracker itself is not alone in the area of lightweight virtual machines, but takes inspiration from Solo [48] and LightVM [49]. Solo [48] takes a low-level approach and divides the hardware into multiple logical partitions, prioritising performance rather than isolation. As the VMM in Solo allows privileged instructions to execute directly on hardware, it is not applicable to multi-tenant cloud computing environments. The other work describes LightVM [49], a redesign of Xen, and achieves a major decrease in boot times, given that the workload is customised for the runtime, such

as a unikernel [50]. A unikernel is a purpose-built kernel for a specific application, in comparison to Linux, which is considered general purpose.

With these works in mind, the model and architecture of Firecracker is more similar to that of traditional VMMs, such as QEMU, however with a limited device model. Firecracker [4] does not support arbitrary operating systems, but has support for both Linux and OSv [51] VMs. OSv is an operating system optimised for VMs, but does run Linux applications.

If we now consider trusted computing in PaaS offerings, efforts that have been made for integrity-verified containers can be applied to solutions based on containerisation, such as gVisor. An example of such work is Container-IMA by Luo *et al.* [52]. The attestation mechanism they propose improves privacy of the container and the underlying host and ensures container isolation. Instead of including support for vTPMs in user space, the mechanism continues on the idea with a shared measurement agent in kernel space, where all application layers are measured by the IMA. The novelty of their work lies in their method for multiplexing PCRs for ephemeral container workloads.

However, for PaaS solutions based on virtualisation, such as Firecracker, no effort has been made to enable trusted or confidential computing. As our work aims to integrate TPMs into Firecracker, already available solutions, such as the mentioned SvTPM [34] implemented on HyperEnclave [35], and Linux IMA [5] can be used to provide trust, confidentiality, and integrity to PaaS offerings.

4
Method

As mentioned in section 1.1, the research question is divided into multiple parts and require specific methods to answer each of them. This chapter describes the purpose of the different components required to answer the research questions, whereas the implementation of the components is described in the next chapter. This chapter begins by describing how a vTPM can be created and later used with Firecracker. Then follows the required changes to Firecracker to support the mentioned vTPM. The chapter then continues with a description of the performance tests that are executed and how they are performed. Finally, the attack surface analysis method used in the project is defined.

4.1 Using TPMs in a Lightweight VMM

Using and running a VM with a TPM requires interaction with multiple processes, as well as creating temporary directories and files. Therefore, a measurement system that manages VMs and vTPMs is required. To answer the research questions, the system also measures and stores data from executions. The tasks performed in the system are the following:

1. Perform the trust establishment procedure, see subsection 2.1.1, and instantiate a new vTPM.
2. Start a VM with a reference to the recently created vTPM.
3. The VM communicates with the vTPM throughout the VMs lifecycle
4. Save the start-up time and memory overhead of the VM to storage

In Figure 4.1, the different tasks can be seen and with which components they interact. These tasks are not specific to Firecracker, and can work with any VMM to measure startup times and memory overhead.

The TPM resource pool, specified in question (2) in section 1.1 also resides in the measurement system, where an algorithm is required to mitigate the exhaustion of the resource pool in the case of ephemeral workloads. The system needs to be able to execute with three different setups: standard Firecracker VM without TPM, modified Firecracker with TPM, and modified Firecracker with pooled TPMs. These setups are used by the performance test suite. For the two setups running with TPMs, a modified Firecracker binary with TPM support is used.

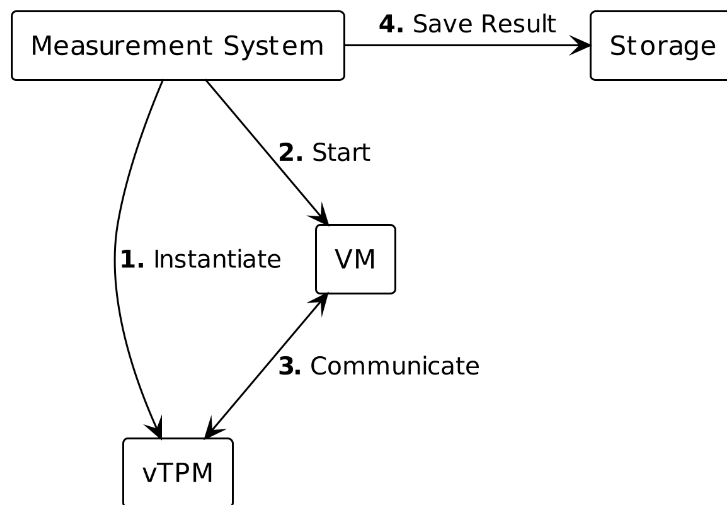


Figure 4.1: The tasks performed by the measurement system and the components it interacts with

4.2 Required VMM Changes

The modified Firecracker binary needs the ability to pass TPM devices to guest VMs. The current external device capabilities of Firecracker are small, focusing on paravirtualised devices, using the virtio standard [19]. Therefore, adding TPM device functionality to Firecracker requires research into virtio and Linux TPM device drivers along with a Rust implementation within Firecracker. However, since other VMMs already support virtual TPM devices [53, 27, 28], they can serve as an inspiration for implementation.

The updated Firecracker device model can be seen in Figure 4.2. The boxes with blue, dotted outlines denote the added components from open-source projects, whereas the boxes with green, dashed outlines are the components implemented in this project. The vTPM started on the host OS is connected to the Firecracker VMM. The implementation of the virtual TPM device in Firecracker exposes the device according to the virtio standard [19] to be consumed by the guest. On the guest side, there is currently no virtio TPM device driver in standard Linux. However, in a Linux fork by Google, such a driver is present. As that driver is open source, it can be used for this purpose [54]. This device driver uses the already existing architecture for TPMs in the Linux kernel and integrates with the IMA.

4.3 Test Configuration

As mentioned, the tests run multiple scenarios with multiple setups. The first setup, *baseline* runs an unmodified Firecracker binary without a TPM, to serve as a comparison value. The second setup, *on demand* performs a trust establishment procedure for a vTPM and starts the vTPM process before starting each Firecracker VM. The third setup, *pool* maintains a resource pool of vTPMs and allocates one of

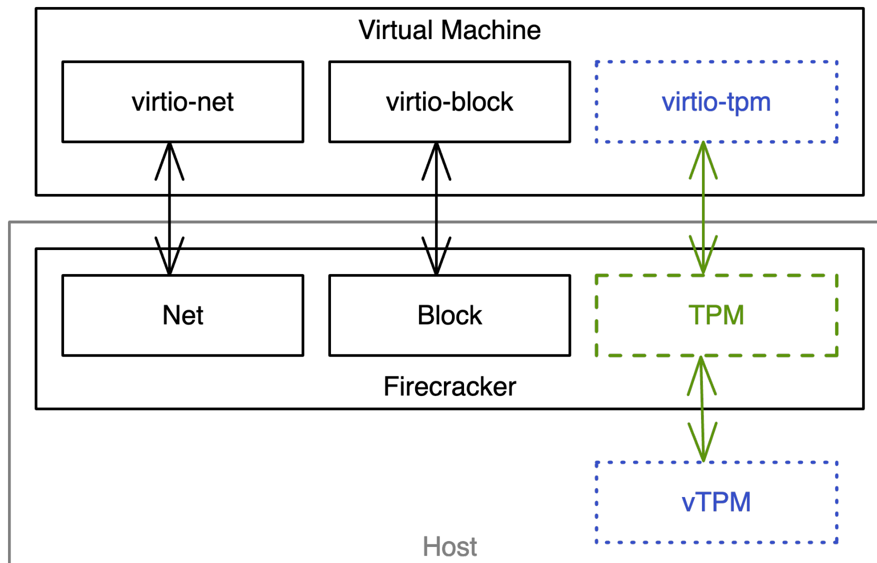


Figure 4.2: The device model of Firecracker with TPM device support

them to each Firecracker VM before starting it.

Regarding the OS used by the VM, a Linux kernel compiled with virtio TPM support along with a simple init script is used. The init script shuts down the VM directly. The reason to not start any actual service is to minimise the number of components that can impact the startup time. The same OS configuration is used for all startup time tests.

The scenarios executed are the following:

- 500 VMs, serially executed
- 1000 VMs, 50 running concurrently

These scenarios are the same as those used in benchmarks made comparing Firecracker with QEMU and Cloud Hypervisor [4]. Based on these scenarios, the results is visualised using a graphed *cumulative distribution function* (CDF) of the startup times. A CDF is used instead of a scatter plot to better visualise the distribution of our results. Along with these graphs, a table of *min*, *max*, *average*, *p95* and *standard deviation* is shown.

In addition to the startup time metric, the measurement system also determines the memory overhead of adding the vTPM to the VM. As the VM needs to be kept running to be able to analyse the memory, it uses a different init program, which spawns a shell. When the memory measurement is finished, the VM is manually terminated from the host. Memory overhead is measured using the *pmap* [55] tool, which displays Linux process memory maps. Only non-shared memory is counted, as it represents how the system scales with more VMs. Linux allocates the same memory slice among processes using the same shared memory, therefore, it does not have an impact on a system with thousands of VMs. The memory overhead is only evaluated with the baseline and the on demand TPM setup. The memory usage of

the Firecracker process and the vTPM is summed for the TPM setup. VMs with different memory sizes are used to check whether the overhead increases together with the memory size. The memory sizes used are 128, 256, 512, 1024, 2048, 3072, 4096, 6144, and 8192 megabytes.

The TPM pool is not considered, as that is not part of either the Firecracker process nor the vTPM. The purpose of the TPM pool implementation in this project is to isolate the boot time from the vTPM startup time. In a real-world scenario, the pool itself would be implemented by the cloud provider to optimise for their use case. Due to this implementation, the pool does not need to be considered for the memory overhead results.

4.4 Attack Surface Analysis

The definition of an attack surface for the project is 'The attack surface is the union of all possible ways an attacker could cause damage to a system' [56, p. 99]. This definition is orientated towards the attacker and can therefore be aligned towards how Firecracker is used in practice and what that means from a security perspective. The attacker in this case is a user of a PaaS offering, which implicitly becomes a user of a Firecracker VM. Therefore, the entire VM is assumed to be malicious. To limit the analysis, the cloud provider is assumed to not be malicious and to have physical security in place that disregards hardware attacks.

In order to answer question (3) in section 1.1, the current virtual device design of Firecracker is investigated and compared with the modification of the Firecracker software needed to incorporate the TPM functionality. The changes introduced are examined from a security perspective, and possible paths and channels for adversaries are analysed.

A quantitative measurement is the change in lines of code introduced by adding the TPM device to Firecracker. The authors of Firecracker describe a correlation between hypervisor code size and attack surface, which motivates this metric [4]. The CLOC tool is used to measure the lines of code changed [57].

The goal of this analysis is to understand whether any of the security-orientated goals in the Firecracker paper [4] are violated by adding the TPM functionality. The security goals they mention are focused on isolation with the underlying purpose of running multiple potentially malicious workloads. Isolation is interesting for the changes introduced by adding TPM support to Firecracker as it does include communication with external processes. Therefore, these changes are compared to other Firecracker components that are capable of communicating outside of the VMM process.

5
Implementation

This chapter describes the implementation of the different components of this project along with a description of the hardware and software environment used. The implementation contains multiple parts, such as the changes made to Firecracker and the TPM pooling algorithm. This chapter also discusses the tests run and all prerequisites needed to start a Firecracker VM with a TPM. The source code for the project can be found at https://github.com/DATX05-CSN-8/fctpm-2023.

5.1 Firecracker TPM Device

The implementation of the TPM device in Firecracker consists of two main components, one communicating with the vTPM, *SWTPMBackend*, and one communicating with the VM, *TPMVirtioDevice*. They can be seen in Figure 5.1. These two components can be seen as the two green arrows from the TPM box inside Firecracker in Figure 4.2. The arrow between the vTPM box and the TPM box is represented by the *SWTPMBackend*, and the arrow between the TPM box and the virtio-tpm box, is represented by the *TPMVirtioDevice*.

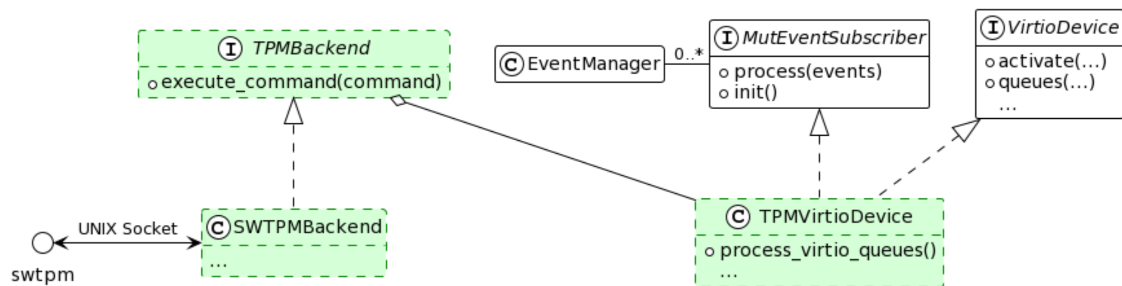


Figure 5.1: An UML class diagram of changes implemented to Firecracker where the boxes with green colour and dashed borders are the additions

The *SWTPMBackend* is communicating with the swtpm over a UNIX domain socket. A UNIX domain socket provides interprocess communication with semantics similar to a network socket. The *SWTPMBackend* was inspired by the vTPM device in the Cloud Hypervisor project [58], as it also uses swtpm over a UNIX domain socket.

To account for platform setup procedures, swtpm exposes a control channel, which is used to both initialise the TPM and to set a file descriptor for client TPM

commands. The file descriptor is one side of another UNIX socket that is created in *SWTPMBackend*. The other side of that socket is used to send and receive messages inside Firecracker.

The *TPMVirtioDevice* implements the *MutEventSubscriber* and *VirtioDevice* which already exists in Firecracker. By using these interfaces, the virtual device fits into the Firecracker device model, and no major changes were needed regarding the internals of Firecracker. Another major advantage of implementing these interfaces was that interrupt registration and device activation are taken care of by already existing components within Firecracker. Firecracker also passes the memory address of the device automatically to the kernel command line parameters for device discovery by the Linux kernel.

In the virtual device implementation, a single queue is used for communication between the VM and Firecracker. Using a single queue was possible as it is always the client who initiates communication in the case of a TPM, which can be compared with a network device where any side might initiate communication. This difference is also apparent when comparing the virtio network device in Firecracker with our virtio TPM device, as the virtio network device uses 2 queues.

A more detailed sequence diagram of the communication between the components related to the use of TPM in Firecracker can be seen in Appendix A.2.

5.2 Test Environment Setup

The hardware environment consists of two different setups. One for the development and functional tests of the project and one for executing the performance test suite. The hardware environment during the development phase was an x86-64 computer with a physical TPM, capable of virtualisation.

In the performance test phase, CloudLab was used. It is a scientific infrastructure that gives bare metal access to cloud computing resources [59]. More specifically, the tests were run on an instance consisting of two Intel Xeon Gold 6142 with 16 cores each and 384 GB of memory.

To verify both hardware environments, two tests were performed. The first test had the purpose of verifying the virtualisation capabilities of the environment. To perform this test, a pre-built Firecracker binary was used along with a Linux distribution created by LinuxKit, a toolkit for building reproducible Linux distributions. The assertion was that the kernel started up correctly. The second test was intended to verify the vTPM functionality in a VM. This test was done using QEMU, swtpm and a Linux distribution created by LinuxKit. The assertion was that the TPM could be interfaced with from the virtual machine.

In addition to the tests, software packages were installed to enable the building of Firecracker and the necessary components.

5.3 Test System Design

An overview of the entire system design can be seen in Figure 5.2. The system contains multiple components and has the ability to instantiate TPMs and Firecracker VMs, it also reports the startup time results to a data store.

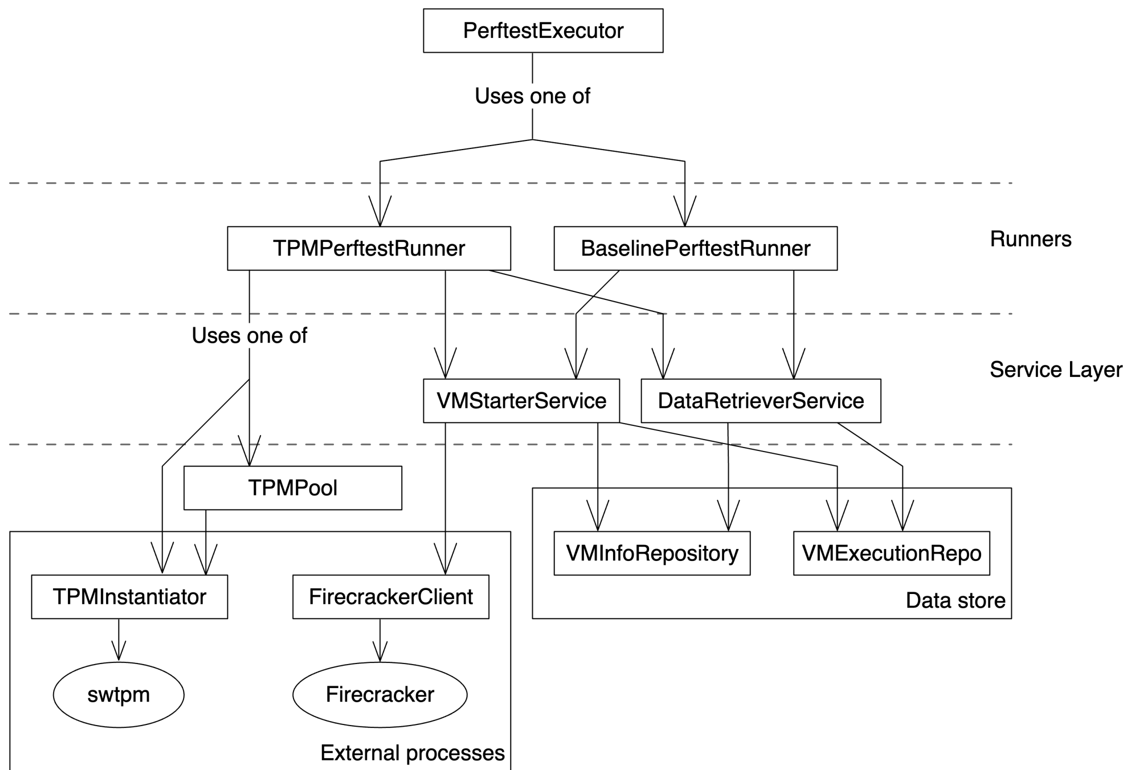


Figure 5.2: Diagram of the test system architecture

The system is implemented in Golang, where all components are contained in the same application. The system uses a layered architecture to separate concerns and maintain reasonable abstraction in all components. An example is the *TPMPerftestRunner*, which uses a simple interface to allocate TPMs, as can be seen in Listing 1. This interface is implemented by both the *TPMInstantiator* and the *TPMPool*, enabling the *TPMPerftestRunner* to depend on an interface without knowledge of the concrete implementation.

```
type tpmallocator interface {
        Allocate() (*tpminstantiator.TpmInstance, error)
        Return(*tpminstantiator.TpmInstance) error
}
```

Listing 1: Interface for the TPM allocator

The system exposes a *command-line interface* with which the user can interact. Based on the command-line arguments, the performance test scenario can be specified to determine whether to test the baseline, on demand, or pooled implementation. Other command-line arguments include which Firecracker binary to use, what kernel and

init should be used for the VMs among others. The help text of the program can be seen in Listing 2.

```
$ ./perftest -help
Usage of perftest:
    -boot-log-path string
        (optional) path to output boot logs
    -clean
        Clean the output database and csv
    -firecracker-bin string
        File path to the firecracker binary that should be used
    -init-path string
        Path to Firecracker init
    -kernel-path string
        Path to Firecracker kernel
    -parallelism int
        The desired VM parallelism
        (default 1)
    -result-path string
        Path to CSV file to create
        (default "output.csv")
    -temp-path string
        Path to temporary data directory
        (default "/tmp/firecracker-perftest")
    -total-vms int
        The total number of VMs to run as part of the perf test scenario.
        (default 20)
    -type string
        Type of performance test to run. Either 'baseline', 'ondemand' or 'pool'
        (default 'baseline')
```

Listing 2: Help text for the performance test command line interface

5.4 Test Suite Tools

The measurement system was designed to record the startup time and memory overhead from within the performance test system. It is done within the same Go application. Goroutines and channels were used to manage parallel operations, where a goroutine is a lightweight thread managed within a Go process and a channel is a thread-safe communication primitive. Measuring the startup time is done by saving the current clock before starting the VM and after the VM has exited to the *VMInfoRepository*. The difference between the two timestamps is considered the startup time for the VM.

To verify that no measurement errors occurred, the internal Firecracker boot timer device is used. The boot timer listens for a write to a specific memory address from the guest, which when written to, makes Firecracker print the time since the process started. The difference between the value printed and the measurements made by the test suite gives an indication of whether any measurement errors occurred.

Regarding memory overhead, the pmap tool is used, as previously mentioned. The tool reports the memory map of a process. An example of the output of the tool can

be seen in Listing 3. As can be seen, the mode for each mapping is displayed, and the non-shared memory consists of each map where the write bit is set. The output is parsed to find the non-shared memory of the VM, and, if used, the attached swtpm.

```
229249:   ./firecracker-tpm --no-api --config-file ...
Address           Kbytes     RSS   Dirty Mode  Mapping
0000000000400000    2008    1816       0 r-x-- firecracker-tpm
00000000007f6000     204     100       4 rw--- firecracker-tpm
0000000000829000       8       8       8 rw---   [ anon ]
0000000000f8a000      40      40      40 rw---   [ anon ]
00007f2e60443000       4       4       0 r--s- Stockholm
00007f2e60444000       4       0       0 -----   [ anon ]
00007f2e60445000       8       0       0 rw---   [ anon ]
00007f2e60447000       8       0       0 -----   [ anon ]
00007f2e60449000    2052      20      20 rw---   [ anon ]
00007f2e6064a000      12       8       8 rw-s-   [ anon ]
00007f2e6064d000       4       4       4 rw-s- zero (deleted)
00007f2e6064e000       4       0       0 -----   [ anon ]
00007f2e6064f000  262144   41328   41328 rw---   [ anon ]
00007f2e7064f000       4       0       0 -----   [ anon ]
00007f2e70650000       4       0       0 -----   [ anon ]
00007f2e70651000      12       4       4 rw---   [ anon ]
00007fffe247e000     132      24      24 rw---   [ stack ]
00007fffe24df000      16       0       0 r----   [ anon ]
00007fffe24e3000       8       4       0 r-x--   [ anon ]
---------------- ------- ------- -------
total kB          266676   43360   41440
```

Listing 3: Example of *pmap* output from a running Firecracker VM

For both the startup time and the memory overhead, the results are saved and later used to create appropriate tables and charts.

5.5 Pooling Algorithm

For the test case of running VMs with TPMs allocated from a pool, the pool adheres to a specific resource pooling algorithm. The purpose of using a pool is to move the resource allocation time from a critical point in time to a time where resources and time are less scarce. Taking into account the use case with vTPMs, a key generation procedure needs to be executed, which incurs a significant allocation time. A characteristic of a vTPM resource is that it cannot be reused, which therefore becomes a property of the resource pool.

The resource pool is passed a resource allocator, which adheres to an interface similar to the *tpmallocator* interface, which can be seen in Listing 1. The more general resource pool interface can be seen in Listing 4. Along with the resource allocator, some configuration may also be passed, depending on the specific pooling implementation.

The algorithm implemented for this project is a simple pre-allocated buffer. It works by receiving a parameter on creation that determines the number of resources to allocate. Before the pool is returned to the caller, one resource is allocated per slot

```
type resourceAllocator[T any] interface {
    Allocate() (*T, error)
    Return(*T) error
}
```

Listing 4: Generic resource allocator interface

in the buffer. To ensure thread safety of the pool, a *mutex* object is used. A mutex object can be locked and unlocked, and when it is locked, it cannot be locked until after it is unlocked. This lock ensures mutual exclusion, which is useful when shared variables are accessed from multiple threads at the same time. The mutex is used when a resource is requested from the pool to ensure that each resource is allocated to one caller only. When a resource is returned to the pool, functionality is delegated to the allocator to deallocate it. A generic implementation of such a pre-allocated resource pool can be seen in Appendix B.

5.6 Starting a Firecracker VM with a TPM

This section describes how to run a Firecracker VM with a TPM, using swtpm. It includes a guide on how to install prerequisites along with building a Firecracker binary from source. Continuing, it describes how to build a Linux kernel image with an accompanying init program to be used with the VM. Furthermore, the section includes how to perform the key generation procedure for a vTPM, how to start the TPM, and ends with how to start the VM.

5.6.1 Prerequisites

The instructions are made for Ubuntu 22.04 running on an x86-64 processor capable of virtualisation with at least 8GB memory. It is assumed that the working directory is the root of the project repository. Furthermore, some packages need to be installed and the current user need to be added to the kvm user group to be able to interact with KVM. To accomplish these installations, the following commands should be run:

```
$ sudo apt-get update
$ sudo apt-get install docker.io make patch
$ sudo usermod -a -G kvm $USER
$ sudo usermod -a -G docker $USER
```

Also, Linuxkit and swtpm needs to be installed by running the following command:

```
$ make -C requirements
$ swtpm --version # Prints the swtpm version if it is available
$ linuxkit version # Prints the linuxkit version if it is available
```

Since the group assignment of the current user has been changed, the terminal session needs to be restarted before the next step.

5.6.2 Compiling Firecracker and Building a Linux Kernel

To be able to run Firecracker with the added TPM functionality, it needs to be compiled from source with the added patches. For convenience, the changes needed are available as patch files in the repository of the project, and the binary can be compiled using the following command:

```
$ make -C modules/firecracker build
$ # Creates a firecracker binary in modules/firecracker/bin/firecracker
```

In order to start a Firecracker VM, a Linux kernel and accompanying init program is also needed. These can be built using the following command:

```
$ INIT_NAME=shell-init make -C vm-image out/fc-image-kernel
$ # Creates kernel and init program at vm-image/out
```

5.6.3 Starting a vTPM and Running a VM

As the Firecracker VM needs the TPM to be available once it starts, the swtpm process needs to be started before the VM. The commands below performs the key generation process for the TPM and then starts the swtpm process with a specified UNIX socket:

```
$ mkdir -p .tmp/tpm
$ swtpm_setup --tpm-state .tmp/tpm \
    --createek --tpm2 --create-ek-cert \
    --create-platform-cert --lock-nvram
$ swtpm socket --tpmstate dir=.tmp/tpm --tpm2 \
    --ctrl type=unixio,path=.tmp/tpm/socket --flags startup-clear
```

As this process needs to be kept running when the VM is started, another terminal is needed to start the Firecracker process. The command to run is the following:

```
$ modules/firecracker/bin/firecracker --no-api \
    --config-file modules/vm-start/shell-config.json
```

If successful, it prints the Linux kernel output and returns the user to a shell inside the VM. Running the following command displays that the TPM is available from within the VM.

```
$ ls -l /dev/tpm0
```

The VM can be stopped by running `exit` from within the VM shell. After that, the swtpm process can be stopped by pressing `ctrl+C`.

6

Results

This chapter presents the test results, mainly the effects of integrating TPMs into Firecracker with regard to performance and security. The data obtained by running the test suite have been parsed, and are presented in diagrams for an easier understanding and overview. Regarding the security aspect, the changes to the attack surface are presented at a technical level.

6.1 Performance overhead

The performance impact of integrating TPMs into Firecracker has been evaluated by measuring the changes in VM startup time and memory overhead. Changes in VM startup time were evaluated with two different scenarios, 500 VMs running without concurrency and 1000 VMs with 50 VMs running concurrently using the performance test suite explained in chapter 5. Measuring the VM startup time was done with 3 different setups: baseline, on demand, and pool. The *baseline* setup uses an unmodified Firecracker binary, whereas the *on demand* setup and the *pool* setup uses a Firecracker binary with added TPM support. The difference between *on demand* and *pool* is that *on demand* allocates TPMs as they are needed, whereas *pool* allocates TPMs before the VMs start. Therefore, the *pool* setup removes the TPM allocation time from the startup time. To ensure that no measurement errors occurred, the startup time measured from the performance test suite was compared with the reported time from the boot timer device contained in Firecracker. The difference between the two was of reasonable size, ranging from $10 - 20\%$, implying the trustworthiness of the results.

The memory overhead results describe the memory usage of each VM, disregarding the memory actually available inside the VM. Three different setups are used here as well; it includes a VM with an unmodified Firecracker binary, as well as two setups with modified Firecracker binaries. The two modified setups differ in whether or not the memory of the swtpm processes is counted.

Scenario 1: 500 VMs, no concurrency

The results of the scenario when running 500 VMs can be seen as a chart of the cumulative distribution in Figure 6.1 and as numerical data in Table 6.1.

An important aspect of the chart and the table is the deviation. In the chart it is

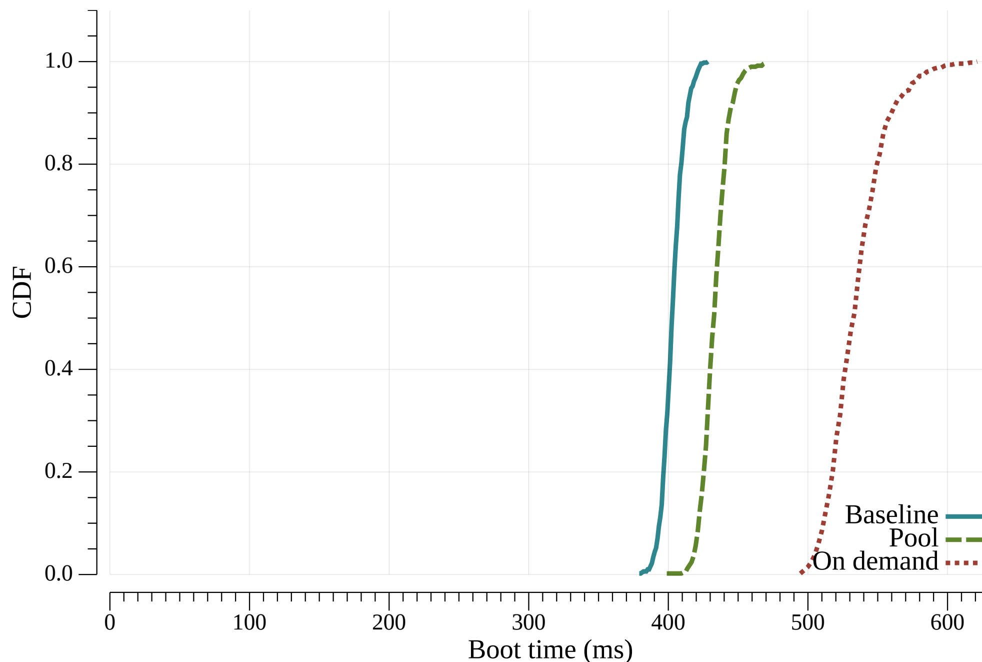


Figure 6.1: Chart illustrating the boot time in milliseconds for 500 VMs serially executed

Type	Min	Max	Average	p95	p95 Change	Std. Deviation
Baseline	379.17	428.47	403.07	416.51	-	7.78
Pool	398.89	471.32	432.99	448.97	+8%	9.40
On demand	494.56	621.25	534.42	572.90	+38%	20.29

Table 6.1: Resulting boot time for 500 VMs serially executed

seen as the width of each CDF, whereas in the table it is can be seen in the *Standard Deviation* column. As can be seen, the change in deviation between the baseline and the pool implementation is significantly smaller in comparison to the change between the baseline and the on demand implementation. With regards to absolute numbers, the average column of the table tells us that the same pattern is seen here. Another interesting metric is the *p95* column of the table, which displays the number of samples that are below 95% of all samples. The p95 value can be used to mitigate the impact of outliers while taking the deviation of samples into account. This metric can be seen as a combination of the absolute value and the deviation, which is also seen in the value of the metric.

In general, the pool implementation performs slightly worse than the baseline; however, if the TPMs are allocated on demand, the results display a significantly worse boot time.

Scenario 2: 1000 VMs, 50 concurrent VMs

Considering the other scenario, with 1000 VMs in total and 50 of them running concurrently, as can be seen in Figure 6.2 and Table 6.2, the result changes. The similarity regarding the CDF of the baseline and the pool implementation is less apparent, and the curve of the pool implementation is slightly wider. However, the same pattern as in Figure 6.1 can be seen by comparing the pool implementation with the on demand implementation. A major change in comparison to the first scenario is that the deviations of all three setups increase by at least three times the previous amount.

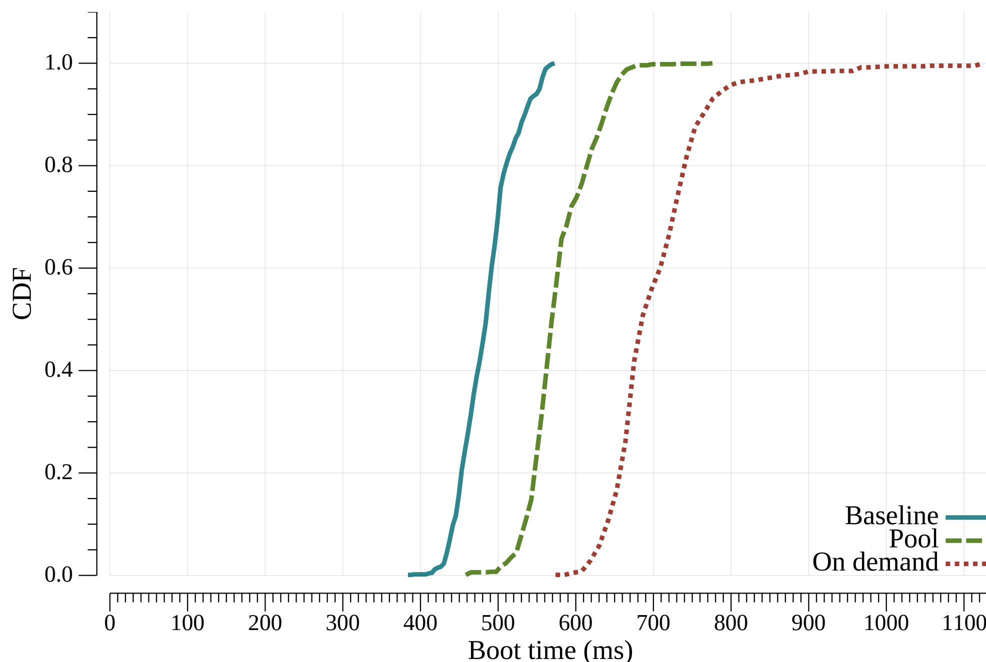


Figure 6.2: Chart illustrating the boot time in milliseconds for 1000 VMs executed with 50 running concurrently

Type	Min	Max	Average	p95	p95 Change	Std. Deviation
Baseline	383.83	572.61	484.54	553.33	-	34.00
Pool	458.44	776.02	577.21	649.83	+17%	39.59
On demand	573.96	1124.99	701.61	790.77	+43%	64.71

Table 6.2: Resulting boot time for 1000 VMs executed with 50 VMs running concurrently

Memory Overhead

The memory overhead test was performed with multiple memory sizes allocated to the VM, to validate that there was no change in overhead dependent on the memory size allocated to the VM. The results did not show any changes in overhead

across VM memory sizes, which can be seen in Appendix A.1. A condensed version, leaving out different VM sizes, can be seen in Table 6.3. As can be seen in the table, the two setups where only the Firecracker process was measured are almost the same. However, when the swtpm process is included, the impact on performance was significantly higher. The increase in memory usage considering only the modified Firecracker process was 4KB, corresponding to an 0.16% increase. However, when memory of the swtpm process was taken into account, the percentage saw an increase of 52.35%.

	KB Overhead	Change
Unmodified	2468	-
Modified excl. TPM	2472	+0.16%
Modified incl. TPM	3760	+52.35%

Table 6.3: Memory overhead of Firecracker and corresponding percentage increase

6.2 Attack Surface Analysis

Language	Unchanged	Changed	Added	Removed	Difference
Rust	61695	2	1257	12	+2.06%
YAML	1183	0	36	0	+3.04%
TOML	426	0	16	0	+3.76%
JSON	30266	0	4	0	+0.01%

Table 6.4: Lines of code edited by adding TPM support to Firecracker

As Firecracker aim to have a minimal implementation, there is relevancy in understanding what impact the modifications have to understand the difference in regards to the attack surface. The difference from the modified Firecracker VMM compared to the original code, in terms of the number of lines of code, has been examined using the command line program CLOC [57]. The scope of the modification made to the Firecracker VMM can be seen in Table 6.4. Considering only Rust files as program code, since YAML, TOML, and JSON is configuration, the changes resulted in an increase of 2% lines of code.

As Firecracker already has general virtio device support there were no change in attack surface with regards to the communication between a guest and the hypervisor code. A reasonable comparison can be made between the virtual network device and the virtual TPM device in Firecracker. The virtual network device creates a simulated network device using the Linux kernel functionality. It is created using a function call to a Linux kernel module and therefore executes kernel code based on the configuration supplied by the user of Firecracker. As the network device is implemented in a kernel module, all network calls from within the guest will also

pass through the kernel. This procedure can be compared with the virtual TPM device that communicates with another userspace process on the host, therefore being significantly less privileged. However, the virtual TPM device uses a UNIX socket to communicate with the swtpm process, and the UNIX socket implementation reside in the kernel. However, this connection is not central for the TPM functionality, but is used as a communication medium.

As the guest communicates with the swtpm process itself, it is considered an extension of the attack surface. Swtpm, being implemented in C, poses a potential target for adversaries to make use of vulnerabilities due to memory safety issues, as can be seen in recent issues [39]. Such issues can lead to code execution vulnerabilities, which in the case of an swtpm process interfaced from a Firecracker VM means a VM escape. A VM escape is when a user within the VM has achieved code execution abilities on the host. However, as the swtpm process runs in the userspace of the host, defence-in-depth principles can be used to isolate the process from the rest of the system.

These findings clearly indicated that the greatest change in the attack surface originates from the swtpm program itself.

7

Discussion

This chapter discusses the results of the experiments, measurements, and analyses performed, both with respect to performance and security. The impact on performance and security introduced by adding TPM support to Firecracker is evaluated in a real-world scenario with CSPs in mind. Additionally, limitations and potential measurement errors are discussed. The chapter ends by describing how recent research within trusted computing can be applied to further improve workload isolation with TPM support in mind.

7.1 Startup Time Evaluation

The results for the test scenario without parallelism, considering only the pooled implementation, show an increase in startup time of 8%, which should be considered acceptable by both CSPs and their customer. As the usage of a TPM in PaaS solutions would most likely be an opt-in feature, similar to that of the current IaaS, it would not be considered an issue for the customer, as they can decide whether to use this functionality or not. However, considering the scenario of running a total of 1000 VMs with 50 VMs running concurrently, the p95 value of the pooled setup resulted in a 17% increase, which is significant.

Also, since the output of the Firecracker internal boot timer device was compared to the values measured by the test suite, measurement errors can be ignored. What can instead be considered is the communication introduced between the Firecracker hypervisor and the vTPM.

7.1.1 Caveats in a Concurrent Environment

The virtual TPM device implemented in Firecracker is synchronous, meaning that the hypervisor wakes up as the guest notifies the device, after that the request is sent to the vTPM, and the guest stays paused for the time it takes to process the request. Also, when sending and receiving data from the SWTPMBackend, the Firecracker process is blocked due to IO calls to the file descriptor communicating with the swtpm process. The IO calls cause the process to go to sleep; meanwhile, another process might start to execute, which can cause a delay. As the number of concurrent processes increases, the risk that the Firecracker process needs to wait before executing again after a blocking call increases.

7.1.2 Differences to Previous Work

When comparing the startup time results in this project with the values in the Firecracker research paper [4], the difference is significant. One potential reason behind this disparity is that the Linux kernel configuration used in this project is different. The reason to use a different kernel configuration is the requirements of the virtio TPM device driver. As the device driver makes use of the complete TPM device subsystem, more kernel components need to be loaded, causing an increase in startup time. An option would be to use a kernel with the same configuration as used in the research paper for the baseline, which would put emphasis on the changes introduced to the Linux kernel and not to the Firecracker VMM itself.

Another reason behind the difference in startup time is the different hardware setups used, mainly the CPU used could have impacted the result. The CPU used by Agache *et al.* [4] is more performant in comparison to the one used in this project. Both single-thread performance and multi-thread performance are significantly lower in this project compared to the one used in the Firecracker research paper. Important for the concurrent scenario is the number of cores, which in the Firecracker paper totalled 48, compared to the 32 that was used in this project. An important aspect regarding the number of cores is the difference between the total number of cores and the desired parallelism. Logically, if the number of cores is greater than the desired parallelism, all workloads can be executed completely in parallel. However, as the number of cores drops in relation to the desired parallelism, the host OS needs to make vital scheduling decisions. These decisions may have caused some workloads to wait, which might have been a reason for the performance difference.

7.1.3 Resource Pool

Regarding the usage of a resource pool, considering real-world use, there would not be a situation where a CSP would allocate vTPMs on demand. Cloud providers go to great lengths to minimise the critical execution path to improve the response times of their PaaS offerings. Therefore, if there is an option to use a resource pool, CSPs have no reason not to use it. However, the resource pool algorithm featured in this work is not suitable for real-world PaaS environments, as it assumes that the total number of virtual machines is known in advance. A viable, although simple, solution would be to always instantiate a new resource and put it in the pool every time an allocation occurs. More advanced solutions could make use of other metrics such as the average time between VM starts and instantiate resources accordingly in a separate thread.

The memory overhead of the resource pool was not considered during the tests, mainly due to the pooling algorithm used. As the pooling algorithm allocates TPMs for the total number of VMs that will be started, it will consume more memory compared to an algorithm that dynamically allocates TPMs. Additionally, the memory usage of a dynamic resource pool is likely to be correlated with the number of resources needed per time unit. The correlation is due to the need to hold a set of not yet allocated resources, a set that grows with the number of resources needed per time unit, which is largely dependent on the deployment method used at a CSP.

7.2 Memory Overhead Evaluation

With regards to the memory overhead comparing the unmodified Firecracker and Firecracker with TPM support, a static increase of 4KB was observed. Such an increase is minor and would most probably be considered tolerable by a CSP. However, including the overhead of the swtpm process, the total overhead increase amounts to approximately 52%. Such an increase might seem major; however, considering the total memory allocation of a VM, it still corresponds to a minor part of the total memory allocated.

7.2.1 Relation to VM Memory Sizes

Since the memory overhead is static across VM memory sizes, the ratio between the overhead and the desired VM memory decreases as the desired VM memory increases. The ratios for the three smallest VM sizes were 2.89%, 1.43%, 0.72%, with the memory sizes 128MB, 256MB and 512MB, respectively. Note that these ratios are for the modified Firecracker VMM, including the swtpm process. The corresponding ratios for the unmodified Firecracker VMM is 1.88%, 0.94%, and 0.47%. These numbers show that, although there is a significant increase in memory overhead, the overhead compared to the VM memory size is still minimal.

7.2.2 Compared to Other VMMs

Another comparison to take into account is the memory overhead results of the Firecracker paper. In their measurements, comparing QEMU, Cloud Hypervisor and Firecracker, Firecracker had the smallest overhead at approximately 3MB, whereas Cloud Hypervisor came second at approximately 13MB, and QEMU last at 131MB. Considering the overhead of the other hypervisors, the increase seen when adding TPM support is negligible.

7.3 Security Evaluation

The result of the attack surface analysis highlights that the major change in the attack surface is seen in the swtpm process. As it is not an option to limit the communication with the swtpm process from the Firecracker hypervisor since it needs to adhere to the TPM specification, other security hardening options have to be used. Assuming the worst, that an adversary can achieve code execution through the TPM commands to the swtpm process, applying defence-in-depth principles come naturally to isolate the adversary and minimise the impact that can be made. Defence-in-depth can be applied and used in multiple ways; one alternative is to isolate the process using Linux kernel primitives, another is to run the vTPM in a separate VM as done with CoCoTPM [36].

7.3.1 Isolating with Linux Kernel Functionality

To harden the security of the swtpm process, Linux cgroups, namespaces, and seccomp filters can be used, among others. Using cgroups can mitigate host resource exhaustion if the adversary floods the swtpm process through the TPM interface. Namespaces improve process isolation by exposing an abstracted view of the network, the process table, and the directory structure to the environment of the swtpm process. Therefore, the use of namespaces can mitigate potential pivots of an adversary, as it can limit the network capability of the process.

By using seccomp filters, the *system calls* (syscalls) a process can perform are limited to a specific set. Syscalls are a way for a userspace process to communicate with the Linux kernel. Using seccomp filters therefore severely limits the ability of a process, as communication primitives within a Linux environment commonly require a syscall. As the vTPM emulates a self-contained hardware component with limited connectivity, few syscalls need to be allowed by the seccomp filters, showcasing the applicability for this use case.

7.3.2 Alternative Isolation Methods

Another option to improve the security of the vTPM implementation is to contain the emulation within the Firecracker program itself. Containing the emulation within Firecracker would, however, increase the program size and memory use of the Firecracker process along with coupling the two components together, which might be undesirable for a CSP. Also, if the swtpm code were to be part of the Firecracker process, an arbitrary code execution exploit would achieve the privilege of the Firecracker process, and therefore an adversary would have more access in comparison to if swtpm were a separate, hardened process.

Containing the vTPM in a separate environment is another option, which would improve isolation even more, at the price of increased overhead. Previous work within the space shows both solutions in separate VMs as well as TEEs [36, 34].

Although the security impact can be interpreted as major due to the risk of vulnerabilities in the vTPM implementation, the time to patch a software component is significantly shorter compared to patching a hardware implementation or firmware. Combining the shortened patch time with the fact that CSPs already use vTPMs in their IaaS offerings implies that the security impact is considered acceptable.

7.4 Practical Application

As Firecracker is one of the many components needed to create a complete PaaS offering, this section tries to describe other components needed to be able to perform trusted computing in a PaaS environment. First, the provider itself must establish a mechanism allowing customers to verify the integrity of the underlying hardware. One option is to use TCCP [41], which describes such a mechanism. Once the machine integrity verification is completed, it can create and sign keys for the vTPM, which are considered trusted as the platform is verified.

As mentioned, a custom resource pool implementation would also be needed for the TPMs to be usable in practice. As all the mentioned components come into place, the one thing that remains now is the supporting services within the Firecracker VM. Mentioned in the Firecracker paper is that they use a Linux kernel inside the VM. Therefore, the IMA subsystem can be used, taking advantage of the TPM for workload integrity measurements. Given that the CSP publishes digests of the underlying components of the guest system, a customer can determine an expected digest to be present. For ease of use, a CSP would expose a remote attestation service alongside the workload invocation entrypoint, that could be used by a customer. It could also be accomplished as part of the workload invocation, through some request metadata. The specific implementation would be determined based on what role the CSP wants to take in their offerings and how much fine-grained access to underlying mechanisms the customer requires.

With the mentioned additions to a PaaS offering, a customer can run their workload, integrity verified, completely transparent, meaning that the customer does not have to do anything more than ticking a box when deploying their service.

7.5 Future work

As this work is aimed at a specific building block which enables trusted computing for PaaS offerings, there are several improvement areas and subfields that would benefit from future research.

One metric that is neither evaluated nor analysed is the number of *VM exits* occurring. A VM exit is when control is transferred from execution within the VM to the hypervisor code, commonly occurring when a VM interacts with an emulated device. As VM exits can be considered an attack vector [60], it would therefore be beneficial to compare the number of VM exits with and without a TPM. VM exits requires significant insight into KVM components to be able to analyse the different reasons behind the exits. Hence, VM exits have not been researched in this work as it deviates from the main purpose of this work.

As mentioned in section 7.4, future research and work is also needed to apply the results of this work in practice. Within practical applications, there is also multiple research subfields to be further worked upon. One such area is trust processes and trust relationships, related to simulated TPM manufacturing and remote attestation. The concepts described in this project need to be adapted to a larger scale to be applicable for CSPs in practice.

Another subfield lies within OS support for trusted execution of workloads, where further research is needed to apply it to PaaS offerings. This work only uses Linux; however, other types of operating systems, such as unikernels similar to OSv could be more applicable. An issue with OSv taking trusted computing into account is that there is no integrity measurement system or TPM support, which is a topic that would benefit from continued research. Also, efforts can be made to implement some trusted late launch functionality, where a VM can boot before it is known what workload it will execute, to later receive the workload and perform integrity

measurements. By using a late launch, the startup time for workloads would likely be able to be lowered even more. Alongside late launch capabilities, more work can be done on improvements to the Linux kernel boot time, as lower boot times would benefit a CSP regardless of whether late launch is used or not.

The vTPM implementation used is quite important with regard to the security of a trusted PaaS offering by a CSP. As mentioned in section 7.3, multiple actions can be taken to further isolate a Linux process to mitigate potential vulnerabilities. One potential research topic is to implement a vTPM in a memory-safe language, effectively protecting it against memory corruption attacks similar to those seen previously with vTPMs [37].

Another topic that has not been considered is snapshots and migrations of VMs. Although previous work has been conducted with other VMMs regarding this topic [61], it falls outside the scope of this thesis and can be considered as potential future work.

7.6 Ethical Considerations

A problematic aspect of the added TPM functionality is that it could attain false security if the user does not have sufficient entry knowledge on how to use it. A TPM alone does not give any trust properties in a system; it needs to be combined with software to give valuable addition. The incorporation of TPMs and the following increase in interaction between virtual machines and the host might bring increased risk and attack surface. However, this attack vector is not a new issue and has been considered negligible by major cloud service providers in the past.

8 Conclusion

This thesis aimed to integrate software TPM support into the Firecracker VMM along with measuring the performance and security impact of the additions. Taking into account the current support for virtio devices in Firecracker and taking inspiration from similar VMM projects, the modifications made to Firecracker resulted in a mere 2% increase in lines of code.

By incorporating the TPM support, the memory overhead of the Firecracker process saw a negligible increase. When including the swtpm process in the memory overhead, the memory overhead increased significantly. Nevertheless, this overhead remains a small percentage of the total memory allocated to a VM, and would, if used in practice, remain an opt-in feature for customers.

The startup time increased significantly when TPMs were allocated on demand; however, when using a resource pool with pre-allocated vTPMs, the startup times were kept at an acceptable level. Yet, there is still potential to further improve these metrics even further by adjusting the resource pool algorithm based on the workload and the specific hardware used.

Regarding the attack surface, there is an increase originating from the extended communication between the VM and external processes. However, considering that other virtio devices are already supported, the extended support for a virtio TPM fits into the Firecracker device model. Therefore, the additional attack vector lie in the actual TPM implementation, which could easily be interchanged and hardened to mitigate security concerns.

In conclusion, the incorporation of a software TPM into the Firecracker VMM does not have a significant performance decrease. Instead, the extended use case, which features additional trust capabilities and assurance, compensates for the slight deterioration in performance.

Bibliography

[1] J. Surbiryala and C. Rong, "Cloud computing: History and overview," in *2019 IEEE Cloud Summit*, pp. 1–7. DOI: 10.1109/CloudSummit47114.2019.00007.

[2] A. Pichan, M. Lazarescu, and S. T. Soh, "Cloud forensics: Technical challenges, solutions and comparative analysis," *Digital investigation*, vol. 13, pp. 38–57, 2015. DOI: 10.1016/j.diin.2015.03.002.

[3] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "VTPM: Virtualizing the trusted platform module," in *Proceedings of the 15th Conference on USENIX Security Symposium*, 2006.

[4] A. Agache *et al.*, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020, pp. 419–434.

[5] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, 2004, pp. 223–238.

[6] C. Mitchell, *Trusted Computing*. Jan. 1, 2005, ISBN: 978-0-86341-525-8. DOI: 10.1049/PBPC006E.

[7] Microsoft. "Windows 11 specs and system requirements." (Mar. 8, 2023), [Online]. Available: https://www.microsoft.com/en-us/windows/windows-11-specifications (visited on 03/08/2023).

[8] W. Arthur, D. Challener, and K. Goldman, *A Practical Guide to TPM 2.0: Using the New Trusted Platform Module in the New Age of Security*. Apress, 2015. DOI: 10.1007/978-1-4302-6584-9.

[9] Trusted Computing Group, "Trusted platform module library part 3: Commands," Nov. 8, 2019.

[10] C. Gebhardt, "Towards trustworthy virtualisation: Improving the trusted virtual infrastructure," Nov. 2011.

[11] Trusted Computing Group, "TCG specification architecture overview 1.4," Aug. 2, 2007.

[12] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not," in *2015 IEEE Trustcom/BigDataSE/ISPA*, pp. 57–64. DOI: 10.1109/Trustcom.2015.357.

[13] P. Jauernig, A. Sadeghi, and E. Stapf, "Trusted execution environments: Properties, applications, and challenges," *IEEE Security & Privacy*, pp. 56–60, Mar. 2020, ISSN: 1558-4046. DOI: 10.1109/MSEC.2019.2947124.

[14] IBM Cloud Team. "A brief history of cloud computing." (Jan. 6, 2017), [Online]. Available: https://www.ibm.com/cloud/blog/cloud-computing-history (visited on 03/15/2023).

[15] S. W. Devine, E. Bugnion, and M. Rosenblum, "Virtualization system including a virtual machine monitor for a computer with a segmented architecture," U.S. Patent 6397242B1, May 28, 2002.

[16] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The linux virtual machine monitor," in *Proceedings of the Linux symposium*, 2007, pp. 225–230.

[17] P. Barham *et al.*, "Xen and the art of virtualization," in *Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP)*, 2003, pp. 164–177, ISBN: 978-1-58113-757-6. DOI: 10.1145/945445.945462.

[18] E. Bugnion, J. Nieh, M. Martonosi, and D. Tsafrir, *Hardware and Software Support for Virtualization*. Morgan & Claypool, 2017, ISBN: 978-1-62705-693-9.

[19] R. Russell, "Virtio: Towards a de-facto standard for virtual i/o devices," *ACM SIGOPS Operating Systems Review*, pp. 95–103, Jul. 1, 2008, ISSN: 0163-5980. DOI: 10.1145/1400097.1400108.

[20] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2005, p. 41.

[21] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pp. 275–287, ISBN: 978-1-59593-636-3. DOI: 10.1145/1272996.1273025.

[22] "Namespaces(7) - linux manual page." (Dec. 18, 2022), [Online]. Available: https://man7.org/linux/man-pages/man7/namespaces.7.html (visited on 05/12/2023).

[23] "Cgroups(7) - linux manual page." (Dec. 18, 2022), [Online]. Available: https://man7.org/linux/man-pages/man7/cgroups.7.html (visited on 05/12/2023).

[24] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux Journal*, 2014, ISSN: 1075-3583. DOI: 10.5555/2600239.2600241.

[25] "What is Podman? — Podman documentation." (2019), [Online]. Available: https://docs.podman.io/en/latest/ (visited on 03/20/2023).

[26] Anjali, T. Caraza-Harter, and M. M. Swift, "Blending containers and virtual machines: A study of Firecracker and gVisor," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2020, pp. 101–113. DOI: 10.1145/3381052.3381315.

[27] *Implement vTPM support - Issue #2343 - cloud-hypervisor/cloud-hypervisor*, https://github.com/cloud-hypervisor/cloud-hypervisor/issues/2343, Accessed: 2022-11-28.

[28] *Crosvm*. [Online]. Available: https://chromium.googlesource.com/crosvm/crosvm/ (visited on 03/16/2023).

[29] M. J. Kavis, *Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS)*. Wiley, 2014, ISBN: 978-1-118-61761-8.

[30] "About Amazon Web Services." (2023), [Online]. Available: https://www.aboutamazon.com/what-we-do/amazon-web-services (visited on 03/16/2023).

[31] Amazon Web Services. "NitroTPM - amazon elastic compute cloud." (2023), [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/nitrotpm.html (visited on 03/20/2023).

[32] M. Zimmerman. "Virtual trusted platform module for shielded VMs: Security in plaintext," Google Cloud Blog. (Aug. 7, 2018), [Online]. Available: https://cloud.google.com/blog/products/identity-security/virtual-trusted-platform-module-for-shielded-vms-security-in-plaintext (visited on 03/20/2023).

[33] Microsoft. "Trusted launch for Azure VMs - Azure virtual machines." (Feb. 14, 2023), [Online]. Available: https://learn.microsoft.com/en-us/azure/virtual-machines/trusted-launch (visited on 03/20/2023).

[34] J. Wang *et al.*, *SvTPM: A secure and efficient vTPM in the cloud*, 2019. DOI: 10.48550/arXiv.1905.08493.

[35] Y. Jia *et al.*, "HyperEnclave: An open and cross-platform trusted execution environment," presented at the 2022 USENIX Annual Technical Conference (ATC), pp. 437–454.

[36] J. Pecholt and S. Wessel, "CoCoTPM: Trusted platform modules for virtual machines in confidential computing environments," in *Proceedings of the 38th Annual Computer Security Applications Conference (ACSAC)*, 2022, pp. 989–998. DOI: 10.1145/3564625.3564648.

[37] MITRE Corporation. "CVE - CVE-2023-1018." (Feb. 24, 2023), [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-1018 (visited on 03/27/2023).

[38] MITRE Corporation. "CVE - CVE-2023-1017." (Feb. 24, 2023), [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-1017 (visited on 03/27/2023).

[39] F. Falcon. "Vulnerabilities in the TPM 2.0 reference implementation code." (Mar. 14, 2023), [Online]. Available: https://blog.quarkslab.com/vulnerabilities-in-the-tpm-20-reference-implementation-code.html (visited on 03/27/2023).

[40] F. A. M. Ibrahim and E. E. Hemayed, "Trusted cloud computing architectures for infrastructure as a service: Survey and systematic literature review," *Computers & Security*, pp. 196–226, 2019. DOI: 10.1016/j.cose.2018.12.014.

[41] N. Santos, K. P. Gummadi, and R. Rodrigues, "Towards trusted cloud computing," in *Workshop on Hot Topics in Cloud Computing (HotCloud)*, USENIX Association, 2009. DOI: 10.5555/1855533.1855536.

[42] D. G. Murray, G. Milos, and S. Hand, "Improving Xen security through disaggregation," in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2008, pp. 151–160, ISBN: 978-1-59593-796-4. DOI: 10.1145/1346256.1346278.

[43] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[44] M. Chae, H. Lee, and K. Lee, “A performance comparison of Linux containers and virtual machines using Docker and KVM,” *Cluster Computing*, pp. 1765–1775, 2019. DOI: 10.1007/s10586-017-1511-2.

[45] S. K. Tesfatsion, C. Klein, and J. Tordsson, “Virtualization techniques compared: Performance, resource, and power usage overheads in clouds,” in *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2018, pp. 145–156. DOI: 10.1145/3184407.3184414.

[46] R. Morabito, J. Kjällman, and M. Komu, “Hypervisors vs. lightweight virtualization: A performance comparison,” in *2015 IEEE International Conference on Cloud Engineering*, pp. 386–393. DOI: 10.1109/IC2E.2015.74.

[47] E. G. Young, P. Zhu, T. Caraza-Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “The true cost of containing: A gvisor case study.,” in *Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2019.

[48] X. Zhang, J. Ma, Y. Miao, Q. Meng, and D. Meng, “Solo: A lightweight virtual machine,” in *2009 IEEE International Symposium on Parallel and Distributed Processing with Applications*, 2009, pp. 129–136. DOI: 10.1109/ISPA.2009.7.

[49] F. Manco *et al.*, “My VM is lighter (and safer) than your container,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 218–233. DOI: 10.1145/3132747.3132763.

[50] A. Madhavapeddy *et al.*, “Unikernels: Library operating systems for the cloud,” *ACM SIGARCH Computer Architecture News*, pp. 461–472, 2013. DOI: 10.1145/2490301.2451167.

[51] A. Kivity *et al.*, “OSv: Optimizing the operating system for virtual machines,” in *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference (ATC)*, 2014, pp. 61–72, ISBN: 978-1-931971-10-2.

[52] 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)*, USENIX Association, 2019, pp. 487–500, ISBN: 978-1-939133-07-6.

[53] *QEMU TPM device — QEMU 7.1.92 documentation*, https://qemu.readthedocs.io/en/latest/specs/tpm.html, Accessed: 2022-11-28.

[54] *CHROMIUM: Device driver for TPM over virtio (1480209) - Gerrit code review*, https://chromium-review.googlesource.com/c/chromiumos/third_party/kernel/+/1480209/2, Accessed: 2023-04-26.

[55] *Pmap(1): Report memory map of process - linux man page*, https://linux.die.net/man/1/pmap, Accessed: 2023-04-25.

[56] C. Theisen, N. Munaiah, M. Al-Zyoud, J. C. Carver, A. Meneely, and L. Williams, “Attack surface definitions: A systematic literature review,” *Information and Software Technology*, pp. 94–103, 2018. DOI: 10.1016/j.infsof.2018.07.008.

[57] AlDanial *et al.*, *AlDanial/cloc: V1.96*, 2022. DOI: 10.5281/zenodo.7455676.

[58] *Cloud Hypervisor*, Apr. 30, 2019. [Online]. Available: https://github.com/cloud-hypervisor/cloud-hypervisor (visited on 03/16/2023).

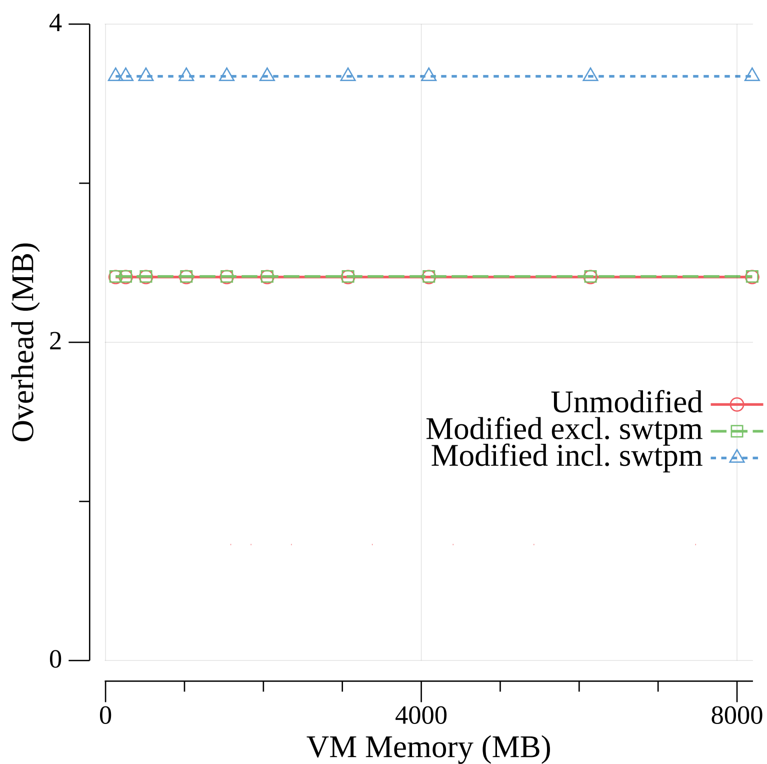
[59] D. Duplyakin *et al.*, “The design and operation of CloudLab,” in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2019, pp. 1–14.

[60] J. Szefer, E. Keller, R. B. Lee, and J. Rexford, "Eliminating the hypervisor attack surface for a more secure cloud," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2023, pp. 401–412, ISBN: 978-1-4503-0948-6. DOI: 10.1145/2046707.2046754.
[61] B. Danev, R. J. Masti, G. O. Karame, and S. Capkun, "Enabling secure VM-VTPM migration in private clouds," in *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, Association for Computing Machinery (ACM), 2011, pp. 187–196, ISBN: 9781450306720. DOI: 10.1145/2076732.2076759.

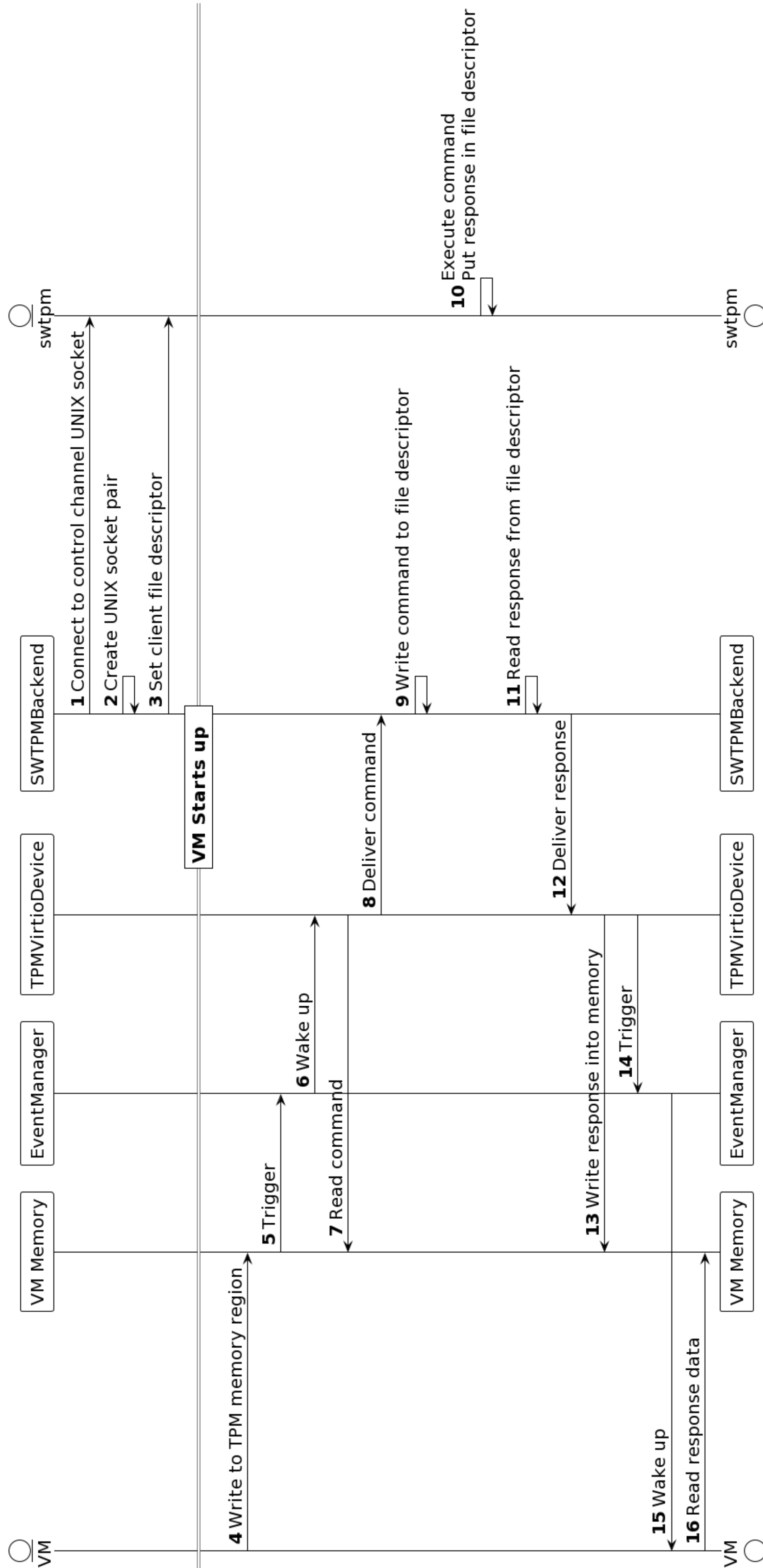
A

Extra Figures

A.1 Memory Overhead Result Chart



A.2 Firecracker TPM Sequence Diagram



B

Pooling Algorithm Implementation

```go
type resourcePool[T any] struct {
    alloc  resourceAllocator[T]
    readyq []*T
    mu     sync.Mutex
}

func NewResourcePool[T any](
    size int,
    allocator resourceAllocator[T]
) (*resourcePool[T], error) {
    t := resourcePool[T]{
        readyq: make([]*T, size),
        alloc:  allocator,
        mu:     sync.Mutex{},
    }
    for i := 0; i < size; i++ {
        inst, err := t.alloc.Allocate()
        if err != nil {
            return nil, err
        }
        t.readyq[i] = inst
    }
    return &t, nil
}

func (p *resourcePool[T]) Allocate() (*T, error) {
    p.mu.Lock()
    defer p.mu.Unlock()
    if len(p.readyq) <= 0 {
        return nil, errors.New(
            "Error occurred, no more elements in tpm pool ready queue"
        )
    }
    inst := p.readyq[0]
    p.readyq = p.readyq[1:]
    return inst, nil
}

func (p *resourcePool[T]) Return(instance *T) error {
    return p.alloc.Return(instance)
}
```