



Degree Project in Cybersecurity

Second cycle, 30 credits

Live Migration of Confidential Virtual Machines

Jakub Ružička

Stockholm 2025

Live Migration of Confidential Virtual Machine

Jakub Ružička

Master's Programme, Cybersecurity, 120 credits
School of Electrical Engineering and Computer Science

English Title

Live Migration of Confidential Virtual Machines

Swedish Title

Live-migrering av konfidentiella virtuella maskiner

Supervisors

Roberto Guanciale

Nicolae Paladi

Examiner

Cyrille Artho

Stockholm 2025

I would like to thank my supervisors Roberto Guanciale and Nicolae Paladi for introducing me to this groundbreaking technology, as well as Canary Bit AB for giving me access to machines that support confidential data processing.

Title: Live Migration of Confidential Virtual Machines

Author: Jakub Růžička

Institute: School of Electrical Engineering and Computer Science

Supervisor: Roberto Guanciale, School of Electrical Engineering and Computer Science

Company Supervisor: Nicolae Paladi, Canary Bit AB

Examiner: Cyrille Artho, School of Electrical Engineering and Computer Science

Abstract:

Given stricter regulations and the transfer of sensitive data to the cloud, there is a clear need to further strengthen cloud security. The latest advances fall under the term confidential computing, which complements existing methods of protecting data during storage and transfer with memory encryption and remote attestation. The introduction of these countermeasures significantly raises the security bar for both remote attackers who operate malicious virtual machines and exploit vulnerabilities in cloud infrastructure, and malicious actors with physical access.

AMD SEV-SNP and Intel TDX are the latest developments implementing confidential computing for server-grade processors. For wider adoption of this technology, effective management of confidential virtual machines, i.e., virtual machines utilizing the protection provided by confidential computing chips, is essential. To facilitate the lifecycle management of confidential virtual machines, the Secure VM Service Module (SVSM) has been introduced as a common layer that can be used across different vendors.

This thesis investigates live migration of confidential virtual machines running under AMD SEV-SNP using the SVSM module. First, the current state of the art is investigated. Since there is no solution for migrating confidential machines with the SVSM module, a migration design is developed and proof of concept is provided for the most time-consuming part of the migration process, random access memory (RAM) migration.

The proposed solution is analyzed and the steps needed to increase its scope and functionality are outlined. A new methodology for evaluating incomplete migration is developed and used to assess the upper limit of the overhead that AMD SEV-SNP confidential machines would represent for the live migration process. Our single-threaded proof-of-concept resulted in a tenfold slowdown in memory page transfers.

Keywords: Confidential virtual machine, Confidential computing, Live migration

Titeln: Live-migrering av konfidentiella virtuella maskiner

Författare: Jakub Růžička

Handledare: Roberto Guanciale, Skolan för elektroteknik och datavetenskap

Företagets handledare Nicolae Paladi, Canary Bit AB

Examinator: Cyrille Artho, Skolan för elektroteknik och datavetenskap

Sammanfattning:

Med strängare regler och överföring av mer känslig data till molnet finns det ett tydligt behov av att ytterligare stärka molnsäkerheten. De senaste framstegen faller under begreppet confidential computing, som kompletterar befintliga metoder för att skydda data under lagring och överföring med minneskryptering och remote attestation. Införandet av dessa motåtgärder höjer säkerhetsnivån avsevärt både för fjärrangripare som använder skadliga virtuella maskiner och utnyttjar sårbarheter i molninfrastrukturen, och för skadliga aktörer med fysisk åtkomst.

AMD SEV-SNP och Intel TDX är de senaste utvecklingen som implementerar confidential computing för serverprocessorer. För en bredare användning av denna teknik är det viktigt att effektivt hantera konfidentiella virtuella maskiner, dvs. virtuella maskiner som utnyttjar det skydd som confidential computing erbjuder. För att underlätta livscykelhanteringen av konfidentiella virtuella maskiner har Secure VM Service Module (SVSM) införts som ett gemensamt lager som kan användas av olika leverantörer.

Denna uppsatts undersöker live-migrering av konfidentiella virtuella maskiner som körs under AMD SEV-SNP med hjälp av SVSM-modulen. Först kartläggs nuläget. Eftersom det inte finns någon lösning för migrering av konfidentiella maskiner med SVSM-modulen, utvecklas ett migreringsförslag och ett proof of concept tillhandahålls för den mest tidskrävande delen av migreringsprocessen, RAM-migrering.

Den föreslagna lösningen analyseras och de steg som krävs för att öka dess omfattning och funktionalitet beskrivs. En ny metod för att utvärdera ofullständig migrering utvecklas och används för att bedöma den övre gränsen för den överflöd som SEV-SNP-konfidentiella maskiner skulle utgöra för live-migreringsprocessen. Vår enkelsträngade proof-of-concept resulterade i en 10x minskning av hastigheten för minnessideöverföringar.

Nyckelord: Konfidentiell virtuell maskin, Confidential computing, Livemigrering

Contents

List of Figures	3
List of Abbreviations	4
1 Introduction	7
1.1 Scenario	8
1.2 Contributions	9
1.3 Ethical, Enviromental and Societal Impact	9
1.4 Outline	10
2 Background	11
2.1 Confidential Computing	11
2.1.1 Trusted Computing Base	12
2.1.2 Threat model	13
2.2 Trusted execution environment	14
2.3 Remote attestation	14
2.4 Live Migration	16
2.5 AMD	18
2.5.1 AMD Platform Security Processor	20
2.5.2 AMD Secure Encrypted Virtualization	20
2.5.3 AMD SEV-Encrypted State	21
2.5.4 AMD SEV Secure Nested Paging	21
2.5.5 AMD SEV-SNP suggested migration	23
2.5.6 Secure VM Service Module	23
2.6 OVMF and IGVM	24
3 Related work	26
3.1 Live migration in QEMU/KVM	26
3.2 Live migration of TEEs	27
3.2.1 Live migration of Intel SGX	28
3.2.2 Intel TDx Migration guide	29
3.2.3 Live migration of AMD SEV-SNP	29
4 Methodology	31
4.1 Research process	31
4.2 Data collection	32
4.2.1 System setup	32
4.2.2 Benchmark setup	33

4.2.3	Workload: stress-ng	34
4.2.4	Workload: Nginx	34
4.3	Software	34
4.3.1	Choice of SVSM	36
4.3.2	Coconut-SVSM overview	36
5	Design and Implementation	38
5.1	Design	38
5.2	Starting the migration handler in SVSM	39
5.3	QEMU and migration handler communication	41
5.4	Source SVSM to destination SVSM communication	42
5.5	Tracking validated pages	43
5.6	Dirty page tracking	44
5.6.1	QEMU guest_memfd interface	44
5.6.2	Possible solutions for confidential guests	45
5.6.3	Security analysis	46
5.7	vCPU stopping	46
5.8	Save zero page optimization	47
5.9	Summary	47
6	Evaluation	49
6.1	Migration speed: stress-ng	49
6.1.1	Evaluation setup	49
6.1.2	Results	49
6.2	Real world scenario: Nginx & Wrk	50
6.2.1	Experimental setup	50
6.2.2	Results	52
6.3	Discussion	52
6.4	Summary	53
7	Conclusion	56
8	Future Work	57
	Bibliography	59

List of Figures

1.1	Confidential computing.	8
2.1	Trusted Computing Base in Confideal Computing	13
2.2	Conceptual data flow during remote attestation.	16
2.3	Live migration	17
2.4	AMD Secure Virtual Machine architecture	19
2.5	Components when running AMD SEV-SNP with SVSM	24
4.1	Experimental setup for a single workload.	35
5.1	Design for live migration of confidential virtual machines.	40
6.1	The total number of pages transferred and the total migration time.	51

List of Abbreviations

- ABI** application binary interface. 23, 43
- AE** Automatic Exits. 21
- AMD SEV** AMD Secure Encrypted Virtualization. 1, 18, 20, 21, 26, 27, 29, 30
- AMD SEV-ES** AMD SEV-Encrypted State. 1, 18, 21, 29
- AMD SEV-SNP** AMD SEV Secure Nested Paging. 1, 3, 9, 12, 18, 21–31, 36, 37, 43, 49, 56, 57
- AMD-PSP** AMD Platform Security Processor. 1, 20, 23, 27, 28, 30, 37, 43
- APIC** Advanced Programmable Interrupt Controller. 30
- Arm CCA** Arm Confidential Compute Architecture. 18, 23
- ASID** address space identifier. 20, 22, 29
- CA** certificate authority. 15
- CCC** confidential Computing Consortium. 11
- CPU** Central Processing Unit. 10, 11, 15–18, 20–22, 28, 46
- CSP** Cloud Service Provider. 8, 15
- CVM** confidential virtual machine. 12, 13, 15, 23, 36–39, 48, 49, 56
- DMA** Direct Memory Access. 13
- DRAM** Dynamic RAM. 20
- GHCB** Guest-Hypervisor Communication Block. 21, 37
- GPA** guest physical address. 22
- GPU** Graphical Processing Unit. 10
- HRoT** Hardware Root of Trust. 12, 14, 15, 20, 27
- IGVM** Independent Guest Virtual Machine. 1, 24, 25, 30, 37, 38, 43

Intel SGX Intel Software Guard Extensions. 1, 27, 28, 30

Intel TDX Intel Trust Domain Extensions. 1, 18, 23, 25, 27–30, 36, 57

IOMMU input-output memory management unit. 22

KVM Kernel-based Virtual Machine. 1, 12, 26, 27, 30, 34, 36, 38, 39, 44, 45, 56

MA Migration Agent. 23, 29

MH Migration Handler. 38, 41, 48

Migration TD Migration Trust Domain. 29

NAE Non-Automatic Exists. 21

OEK Offline Encryption Key. 23

OS Operating system. 13, 23, 24, 28, 29, 41, 43, 44

OVMF Open Virtual Machine Firmware. 1, 24, 25, 29, 30, 38, 44

PCR Platfrom configuration Register. 12

QEMU Quick Emulator. 1, 2, 12, 26, 27, 29, 30, 32–34, 36–39, 41, 42, 44, 45, 47, 48, 52, 55, 56

QMP QEMU’s Machine Protocol. 32, 33

RAM Random-access memory. 9, 10, 16, 26, 27, 30, 32–34, 36, 38, 39, 42, 47, 50, 52, 53, 56

RFC Request for Comments. 29, 30

RMP Reverse Map Table. 22, 27, 37, 43, 45

SEAM Secure Arbitration Mode. 28

SLA Service-level agreement. 7

SoC System on Chip. 11, 14, 15

SPA system physical address. 22

SRAM Static RAM. 20

SSH Secure Shell. 32, 33

SVM Secure Virtual Machine. 3, 18, 20, 21

SVSM Secure VM Service Module. 1–3, 9, 10, 18, 22–24, 27–30, 32–34, 36–39, 41–48, 52, 56, 57

TCB Trusted Computing Base. 9, 12, 13, 23, 28

TDVF Trusted Domain Virtual Firmware. 25

TDx Trust Domain Extensions. 28, 29

TEE Trusted Execution Environment. 1, 8, 9, 11–15, 20, 27, 28

TLB translation lookaside buffer. 18

TLS Transport Layer Security. 11

TPM Trusted Platform Module. 20, 23

vCPU virtual CPU. 2, 26, 29, 30, 33, 38, 39, 41, 46, 48, 56, 57

VM virtual machine. 8, 12–18, 28–30, 38, 41, 57, 58

VMCB Virtual Machine Control Block. 18, 19, 21

VMM Virtual Machine Manager. 21

VMPL virtual machine privilege level. 21–24, 27, 30, 37, 43

VMSA Virtual Machine Save Area. 21, 39

vTPM virtual Trusted Platform Module. 23, 37, 39

1. Introduction

The increasing demand for efficient and cost-effective computing has led many companies and government authorities to migrate their workloads to the cloud, driven by the promise of enhanced scalability, on-demand computation power, and reduced costs. This shift towards cloud computing has transformed the way organizations approach data processing, storage, and management, enabling them to focus on core business activities while leveraging the expertise and infrastructure of cloud service providers. As a result, cloud computing has become a vital component of modern computing, allowing businesses to adapt quickly to changing market conditions and capitalize on new opportunities.

Cloud computing entails the execution of computationally intensive programs on hardware operated by a cloud service provider, which possesses greater computational resources than required and leases the surplus capacity to other users. A prevalent method for deploying these resource-intensive programs to the cloud is through the utilization of virtual machines. A virtual machine is a compute resource that uses software instead of running directly on the physical computer. This software enables multiple virtual machines to run on a single physical computer.

The transition to the public cloud changes the threat model and presents new risk and compliance requirements. The risks include increased attack surface due to shared infrastructure (multitenancy) in cloud environments and direct administrator inspection.

Confidential computing, see Figure 1.1, addresses these challenges by complementing the at-rest and in-transit encryption with encryption of data in-use and proving to the user that data are encrypted on the remotely running virtual machine. The chip manufacturers represent the root-of-trust for this proof. Even though the technology is not without vulnerabilities and limitations, it raises the security significantly, by providing defense-in-depth.

At the same time, cloud service providers offer specific guarantees to their customers, including [Service-level agreements \(SLAs\)](#) that define the maximum time a customer cannot access their virtual machine, ensuring a certain level of availability and reliability for cloud-based services.

To fulfill these commitments, cloud service providers must perform migrations of virtual machines. In scenarios where customer machines operating within a particular data center are nearing the limits of their hardware capacity, it may be necessary to relocate some machines to another data center with lower utilization levels. A different scenario arises when a virtual machine is operating on failing hardware and requires migration to new hardware. To minimize prolonged outages, cloud service providers employ live migration techniques, wherein the majority of

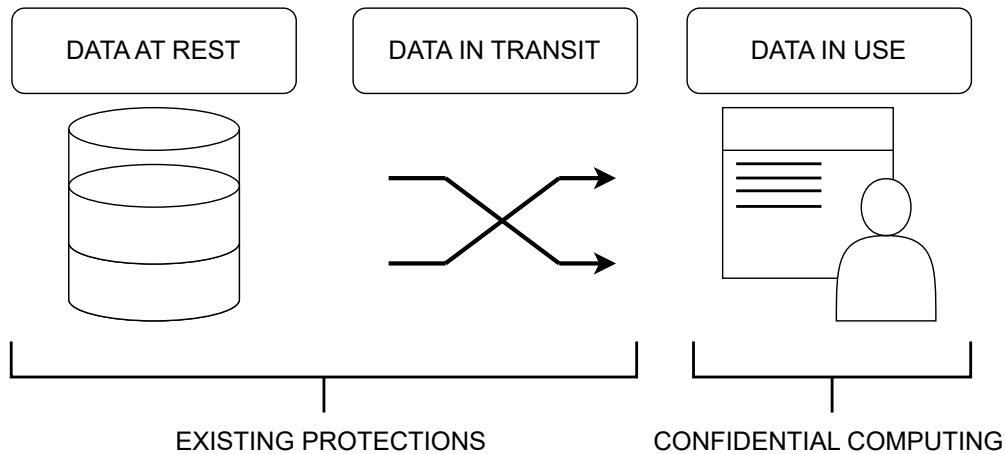


Figure 1.1: Figure depicting the additional protection for data in use provided by confidential computing.

the machine’s data is transferred in real-time while the virtual machine remains operational, either at its original physical location or on the target hardware.

1.1 Scenario

The scope of this thesis focuses on the “cloud” scenario, where a hypervisor runs on a bare metal machine and hosts multiple [virtual machines \(VMs\)](#). There are several actors to consider: (1) [Cloud Service Provider \(CSP\)](#), which operates all infrastructure, has physical access to all machines, and controls all platform software (operating system, hypervisor) except for trusted firmware (e.g., Amazon for Amazon Web Services, Microsoft for Azure, Google for Google Cloud) (2) the cloud customer, who deploys potentially sensitive workloads to execute in [VMs](#) hosted by the [CSP](#) (3) the chip manufacturer, trusted by the cloud customer, which provides hardware and trusted firmware (e.g., AMD, Arm, Intel) and (4) the adversary, who aims to break the security properties of the [Trusted Execution Environment \(TEE\)](#).

The cloud customer wants to start a [virtual machine](#) on the [CSP](#)’s infrastructure and, once the [VM](#) is running, deploy a sensitive workload on that [VM](#). By using confidential computing, the cloud customer adds a new defence against potential threats, specifically vulnerabilities in the [CSP](#)-provided firmware or software that could be exploited by other (malicious) cloud customers. The cloud customer needs to verify that the uploaded [VM](#) has not been tampered with. Remote attestation is used to prove to the cloud customer that the deployed machine is

indeed the one that the user uploaded and that it is running inside a genuine TEE. The chain of trust for this verification is rooted in the chip manufacturer, which provides a secure foundation for establishing trust. By minimizing the Trusted Computing Base (TCB), the security of the system is enhanced. More on trust and how it is established is discussed in Section 2.3.

1.2 Contributions

This thesis investigates the feasibility of live migration using Secure VM Service Module (SVSM) for AMD SEV Secure Nested Paging (AMD SEV-SNP), with a focus on addressing the research question:

What is missing for live migration to be implemented for AMD SEV-SNP?

This entails the following questions:

- (R1) What are the key steps involved in live migration of confidential machines?
- (R2) What essential components and mechanisms must be implemented to enable the designed live migration process, and how do they interact to ensure secure and efficient migration?
- (R3) What are the expected performance overheads associated with live migration, specifically in terms of RAM migration speed and maximum allowable downtime?

Specifically, the analysis of the gaps that need to be addressed for live migration to be implemented provides a comprehensive understanding of the technical requirements and challenges associated with live migration in a confidential computing context (R1). We identify the key components required to enable live migration of confidential virtual machines (R2). We estimate the upper bound of overhead costs in our experimental setup, providing an initial point of reference (R3). The findings of this research inform the development of secure and efficient live migration mechanisms for confidential computing, ultimately facilitating the broader adoption of this technology.

1.3 Ethical, Environmental and Societal Impact

The thesis contributes to research on confidential computing, a field that necessitates consideration of its ethics, sustainability, and societal impact.

Confidential computing has significant ethical implications, particularly in regards to data protection and privacy. By enabling data to be processed in a secure

and isolated environment, confidential computing helps prevent unauthorized access to sensitive information, thereby reducing the risk of data breaches and cyber attacks. This technology also promotes trust in data sharing and collaboration, as individuals and organizations can be assured that their data is being handled in a secure and confidential manner. Furthermore, confidential computing can facilitate the use of sensitive data for beneficial purposes, such as medical research or financial inclusion, while maintaining the confidentiality and integrity of that data. Overall, confidential computing has the potential to promote a more secure and trustworthy digital environment, which is essential for fostering innovation and protecting data in our digitized society.

For a positive societal impact, it is necessary to provide customers with realistic expectations regarding the provided guarantees and risks. Contrary to the often presented view, Confidential Computing does not provide the same guarantees as on-premises computation, and customers should be provided with the necessary information before uploading their workloads.

Regarding sustainability, supporting Confidential Computing requires the replacement of older machines with new ones equipped with specialized hardware. Currently, this primarily concerns [Central Processing Units \(CPUs\)](#), but it is likely to extend to [Graphical Processing Units \(GPUs\)](#) and potentially other peripherals, such as networking cards. The environmental impact of this transition is currently limited due to the relatively limited adoption of Confidential Computing-enabled machines. Nevertheless, as this technology becomes more widespread, its sustainability implications may require careful consideration.

1.4 Outline

Following the introduction, this thesis is organized into five chapters. [Chapter 2](#) and [Chapter 3](#) presents the technologies underpinning confidential computing and live migration, providing a foundation for the research. [Chapter 4](#) describes the applied methodology, outlining the approach and techniques used. In [Chapter 5](#), the research questions (R1) and (R2) are addressed. Design for live migration of confidential machines under [SVSM](#) is proposed, including details of the necessary components and notes on which ones have been implemented. [Chapter 6](#) answers (R3) by presenting the results of the evaluation of the implemented solution for [RAM](#) migration. The thesis concludes in [Chapter 7](#), summarizing the key findings and contributions. Finally, [Chapter 8](#) explores potential future directions and ways to build upon this research.

2. Background

This chapter serves as an introduction to the concept of confidential computing, beginning with a discussion of general principles before delving into the specific technologies addressed in this thesis. Initially, the chapter explores the threat model associated with confidential computing, along with two foundational elements: the [Trusted Execution Environment \(TEE\)](#) and remote attestation. Following this, an overview of live migration is presented. The latter sections examine the evolution of AMD’s technologies that support confidential computing, as well as the file formats employed in this context.

2.1 Confidential Computing

Confidential computing [1] is the protection of data in use by performing computation in a hardware-based, attested trusted execution environment [2]. This is the definition adopted by the [Confidential Computing Consortium \(CCC\)](#). The CCC is a community focused on projects securing data in use and accelerating the adoption of confidential computing through open collaboration. The members include major chip providers (AMD, Intel, Arm) as well as technological giants (Google, Microsoft, Meta Platforms, Nvidia).

We present the terminology used in the definition. The term “protection of data in use” refers to the fact that this protection model concentrates on data that is actively being processed. This data resides in memory and is handled by the CPU, i.e., if not adequately safeguarded, it can potentially be accessed via the data buses connecting the CPU to the memory or within the CPU’s registers and caches. This type of protection serves as a complement to the security measures for data that is stored (data at rest) and data that is being transmitted (data in transit). In contemporary data security practices, the protection of data at rest can effectively be achieved through encryption, while data in transit is protected using cryptographic protocols such as TLS. However, the challenge of securing data during active use remains an area of ongoing research. Various methodologies are being explored in this context, including (1) homomorphic encryption, (2) multi-party computation, and (3) confidential computing [3][4].

The term “trusted execution environment” refers to a secure area of the main processor that protects the data with respect to confidentiality and integrity. The three must-have properties for TEEs are (1) data confidentiality, (2) data integrity and (3) code integrity. There is an emphasis on the TEE being hardware-based, i.e., that chip providers design the [System on Chip \(SoC\)](#) to support enforcement

of the required properties. This also places the chip providers in the position of a root of trust.

Hardware-based [TEEs](#) are particularly well-suited for the scenario under consideration, as a lower root of trust reduces the number of components that require trust. Consequently, this effectively minimizes the size of the [TCB](#). In comparison to other technologies designed for safeguarding data in use, hardware-based [Trusted Execution Environments \(TEEs\)](#) currently offer significantly greater performance than homomorphic encryption and facilitate the seamless integration of existing workflows relative to both homomorphic encryption and multi-party computation.

Confidential computing can establish the trust boundary at varied levels of data isolation, such as [virtual machine](#) isolation or process isolation. For the purposes of this study, we define a [confidential virtual machine \(CVM\)](#) as any [virtual machine](#) that fulfils the following two properties:

- (1) Isolation: encryption and integrity of memory and execution state.
- (2) Remote Attestation: cryptographic measurement of machine state signed with a key that is connected to the chip provider by a chain of trust.

2.1.1 Trusted Computing Base

The virtualization stack depicted in the [Figure 2.1](#) comprises several layers that interact to manage both confidential and legacy [VMs](#). Chip Provider Hardware and Firmware ([AMD SEV-SNP](#)), which offer critical security features to ensure data integrity and encryption. Above this, the firmware (BIOS/UEFI) initializes hardware components and loads the hypervisor [KVM](#), enabling resource management for virtual machines. The hypervisor allows multiple [VMs](#) to run simultaneously on a single host, efficiently coordinating their operations. The Cloud Management Software ([QEMU](#)) then manages the lifecycle and resource allocation of [VMs](#).

The trusted (green) components form the trusted computing base. In contrast, untrusted (red) components are intended to be malicious and may collaborate with other untrusted components to compromise the system. This is realistic because all untrusted components can be managed by a single entity.

The chip provider’s hardware and firmware implements [TEE](#) and establishes the [Hardware Root of Trust \(HROt\)](#) for measured boot and remote attestation purposes. The [HROt](#) is typically implemented as a trusted platform module [5] that stores a set of non-removable private keys sealed directly in the chip during manufacturing and a set of [Platform configuration Register \(PCR\)](#) that cannot be written directly but can be “extended” by a hashing process and therefore contain a cumulative hash of stored values, e.g., system state.

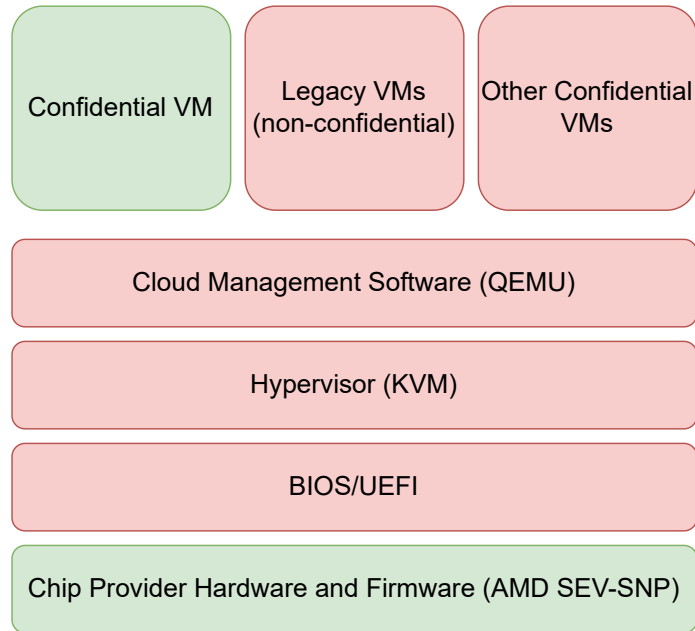


Figure 2.1: Figure showing the components considered trusted (green) and untrusted (red) from the perspective of a single confidential virtual machine. The trusted components compose the **Trusted Computing Base**.

The **TCB** includes the entire virtual machine, including its **Operating system (OS)** kernel and all applications. In a typical situation where Linux or Windows is running inside the virtual machine, the size of **TCB** is significant.

2.1.2 Threat model

Confidential computing assumes an adversary with full control over the underlying hardware, the ability to observe all data entering and leaving the **TEE**, and unrestricted access to system memory, including the ability to perform **Direct Memory Access (DMA)** for both reading and writing.

More specifically, the adversary is assumed to have complete control over the platform’s software stack. This means that it can (1) execute arbitrary code with elevated privileges, (2) start, pause, or terminate confidential **virtual machine** instances at will, (3) monitor and manipulate all network traffic, and (4) fully manage untrusted components, including memory mapping, I/O devices, and processor scheduling. In addition, the threat model assumes that an attacker can gain physical access to the system, allowing them to exploit vulnerabilities in hardware com-

ponents outside of SoC through physical attacks such as cold-boot, bus-snooping attacks, cache monitoring, or plugging devices into existing ports [6].

As usual, an adversary is unable to subvert properly implemented cryptographic primitives and is generally unable to subvert the security guarantees that TEE claims to provide.

However, certain attacks are explicitly not addressed. These include sophisticated physical attacks, which require long-term and/or invasive access to hardware, and availability attacks, as the cloud provider may not allow the VM to run on a machine, i.e., schedule it on a processor, since scheduling is a necessary part of the hypervisor’s capabilities to perform its role effectively and to prevent a misbehaving VM from performing an availability attack on other machines.

2.2 Trusted execution environment

Trusted Execution Environment (TEE) is a broad term used in various contexts. The focus of this work are hardware-based, as this is in the definition of trusted computing, server-side TEEs in general-purpose processors used in conjunction with virtualization technology and with support for remote attestation, as this corresponds to the considered cloud computing scenario.

There are three defining properties for TEEs: (1) Data confidentiality — data processed inside the trusted execution environment are not visible to any party outside of this environment, (2) Data integrity — unauthorized entities cannot add, remove or alter data while in use within the TEE and (3) Code integrity — unauthorized entities cannot add, remove or alter data while in use within the TEE. Data and code are often referred to as a workload, so the last two points can be merged as workload integrity [2][7].

2.3 Remote attestation

The mere existence of a Trusted Execution Environment is insufficient if an untrusted cloud provider can modify the VM during deployment or deceive the guest owner into believing that the VM is operating within a genuine TEE when it is not. Consequently, the primary objective of remote attestation is to establish trust in the state and identity of the remote machine, specifically ensuring that the remote machine is indeed running within a TEE and that the Hardware Root of Trust has not been compromised.

The remote attestation process can be delineated as follows: one participant, referred to as the attester, generates credible information about itself—termed evidence—to enable a remote participant, known as the relying party, to determine

whether to regard the attester as a trustworthy entity. This process is further facilitated by an essential third party, the verifier.

The general flow of information, as illustrated in Figure 2.2, is adapted from the remote attestation specification [8]. Initially, the guest owner measures the **confidential virtual machine** prior to its upload to the cloud, recording this as the **VM image measurement**. The code residing within the **TEE**, specifically the chip provider firmware, acts as the attester, collecting claims about the machine. This includes measuring the uploaded image, proving its authenticity as a genuine hardware-based **TEE**, and incorporating the nonce provided by the verifier to demonstrate the freshness of the evidence.

In this context, evidence constitutes a cryptographic measurement (e.g., a cumulative hash function) of the machine’s state, which encompasses the contents of memory pages, **CPU** state, configuration, and firmware version. This evidence is signed by the **Hardware Root of Trust**, which comprises a set of private keys embedded directly into the **System on Chip** during manufacturing. The chip provider is responsible for publishing the corresponding public keys. Components that require endorsement, as no evidence is generated about them, are referred to as the *root of trust* (e.g., public keys, certificates).

The cryptographic measurement occurs at the machine’s launch, as it is in a predefined state at that moment. When attestation is requested during the runtime of a **VM**, the measurement at launch time is always provided, potentially accompanied by additional data as specified by the verifier. However, if the user is unable to supply supplementary data for inclusion in the measurement, this scheme presents a significant vulnerability regarding freshness and is susceptible to replay attacks. In such scenarios, the **Cloud Service Provider** may store the signed measurement and, if the user attempts to deploy a second **VM** with the same image, the **CSP** could return the previously recorded signed measurement.

The chip provider firmware must be immutable from the cloud provider’s perspective; any attempt to update it should result in the destruction of the secret contained within, as this secret serves as the **HRoT**. The **HRoT** can be verified through a chain of trust rooted in the hardware manufacturer. Thus, the chip provider functions as an endorser, validating the attester’s signed evidence. Typically, the chip provider employs a **certificate authority (CA)** to sign intermediate keys, which subsequently sign machine-specific keys.

Upon signing the evidence, the attester can transmit this evidence to the verifier. The verifier evaluates the evidence according to appraisal policies and generates attestation results to assist the relying party in its decision-making process.

However, critical components remain unspecified, particularly the verifier and the verifier owner. The verifier must be trusted by the relying party, as it is responsible for validating the evidence. Currently, the verifier is often provided

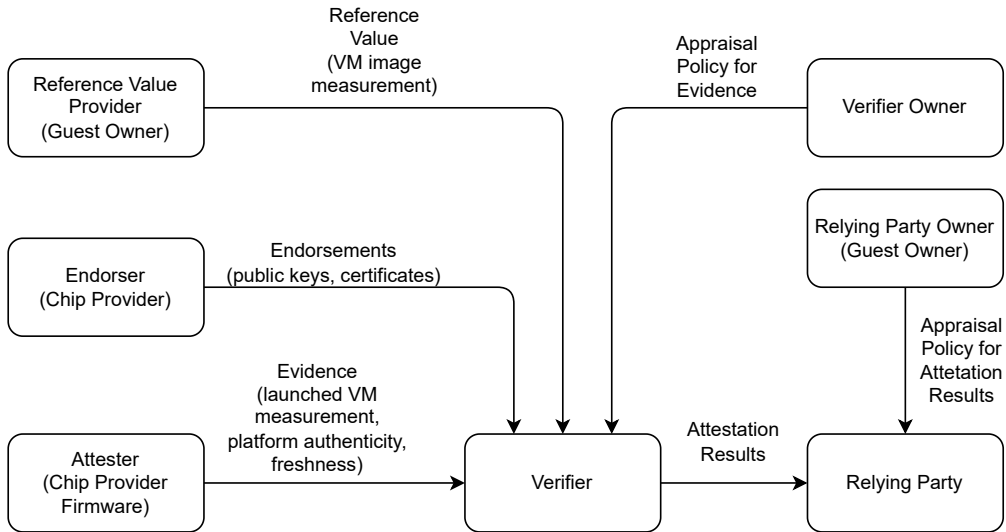


Figure 2.2: Conceptual data flow during the remote attestation, with the mapping to the confidential computing scenario in the brackets, where possible.

by the cloud provider, which is inherently problematic due to the lack of trust in the cloud provider [9]. A more favorable approach would involve the evidence being supplied to the relying party, which could then utilize its own (potentially open-sourced) verifier to ensure trustworthiness and integrity in the attestation process.

2.4 Live Migration

Live migration, as shown in Figure 2.3, is the process of transferring a VM from the host currently running on, called the source, to another host where it should continue to run, called the destination. The goal of live migration is to ensure that the process occurs seamlessly, as if the VM had never been halted.

When live-migrating the machine, there are three states we need to transfer from the source machine to the destination machine: (1) Execution state — RAM, CPU registers, device registers, (2) Storage — disk storage, (3) Connectivity — internet connection, devices. There exist two main migration strategies: pre-copy and post-copy [10][11].

The pre-copy approach is divided into two phases: the push phase and the stop-and-copy phase. In the push phase, the memory pages and possibly other large objects are copied from the source to the destination. In order to transfer the memory efficiently, the push phase is divided into rounds, and the hypervisor

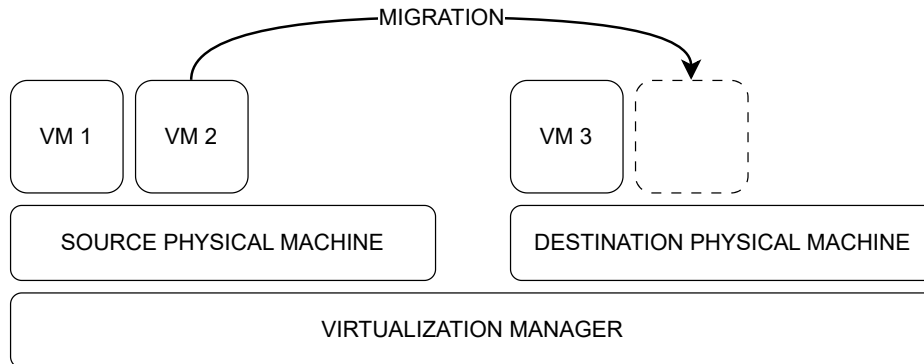


Figure 2.3: Live migration is the process of transferring the state of a guest **VM** from a source to a destination as if the **VM** had never been halted.

manages the bitmap. In the first round, every memory page is transferred, while in any subsequent round, only pages that were written to since the last copy operation are sent to the destination. When a specific condition is met, e.g., the number of rounds, the duration of the push phase or the number of to-be-copied pages is low enough, the machine enters the stop-and-copy phase. In this phase, the source machine execution is stopped, and the rest of the memory pages, along with the execution state, are transferred to the destination machine. As soon as the destination physical machine loads the execution state, the machine is resumed [10].

The second strategy starts with the stop-and-copy phase. The source virtual machine is suspended, and the minimal execution state (**CPU** state, registers, non-pageable memory) is transferred to the destination, where the machine is resumed. Once the virtual machine starts on the destination, the pull phase is entered. In this phase, the rest of the state is pulled from the source physical machine. If a not-yet-copied page should be accessed, the hypervisor generates a network fault to prioritize the transfer of that page [11].

Both approaches may be tweaked in a number of ways in order to optimize different metrics, e.g., total downtime, amount of migrated data, and total migration time required by the specific scenario. All these options, along with ever-increasing virtual machine sizes and strict service level requirements, make the live migration of machines a highly non-trivial task [12][13][14].

This already complex process is further complicated when confidential virtual machines are involved. Efficient migration requires guest cooperation; both source and destination should attest to their counterpart, and guest-defined migration policies must be enforced. The new trust model also comes with new types of attacks, that must be taken into account.

In a fork attack, also called a cloning attack, the adversary’s objective is to create two or more copies of the same confidential machine with an inconsistent state, potentially on different physical machines, in order to undermine some security guarantees [15]. The hypervisor may also decide not to send certain pages, potentially returning the VM to a previous state.

2.5 AMD

The AMD’s **Secure Virtual Machine (SVM)**, marketed as AMD-V, is a set of hardware extensions designed to enable efficient and secure management of virtual machines. We start with the base SVM architecture and then present the individual technologies that have been added throughout the years, namely **AMD Secure Encrypted Virtualization (AMD SEV)** [16], **AMD SEV-Encrypted State (AMD SEV-ES)** [17], and **AMD SEV Secure Nested Paging (AMD SEV-SNP)** [18]. To help manage the confidential guests under **AMD SEV Secure Nested Paging (AMD SEV-SNP)**, the **Secure VM Service Module** was specified. This module has the potential to serve as a common confidential layer for not only **AMD SEV**, but also other confidential technologies, such as **Intel TDX** and **Arm CCA**.

The base SVM introduces a mechanism for a fast world switch between the hypervisor and guest execution contexts, a tagged TLB to reduce virtualization overhead and external memory protection, assists with interrupt handling, virtual interrupt support, and the ability to intercept selected instructions or events in the guest. To facilitate execution, a **Virtual Machine Control Block (VMCB)** 4 KB structure is associated with each guest. The VMCB contains a list of instructions and events to intercept, as well as the specification of the guest execution environment and guest processor state.

Hypervisor initiates the world switch by calling the *VMRUN* instruction with a physical address of the VMCB as the sole argument. The CPU saves the hypervisor state and executes the guest code until the specified instruction or event triggers the *#VMEXIT*. This causes the world to switch back to the hypervisor. Before returning control to the hypervisor, the CPU saves the guest’s state to the VMCB and sets the fields in VMCB to indicate the reason for the *#VMEXIT*. Based on the reason for the *#VMEXIT*, the hypervisor performs a specific action, potentially updating the guest state stored in VMCB, e.g., providing a value read from a specific register. Once completed, the hypervisor executes *VMRUN* to let the guest run again. The loop, as shown in Figure 2.4, runs until the guest code finishes.

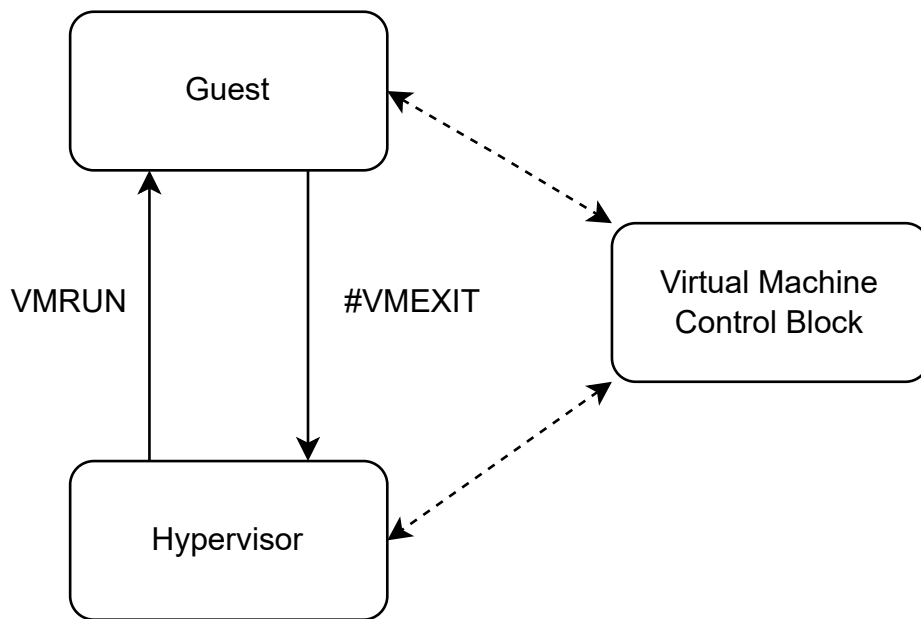


Figure 2.4: In the SVM architecture, the guest execution is initiated by the *VMRUN* instruction. Conversely, the *#VMEXIT* mechanism is used to switch the world back to the hypervisor. The **Virtual Machine Control Block** contains information used to manage the guest execution context.

2.5.1 AMD Platform Security Processor

The AMD Platform Security Processor (AMD-PSP) is a security coprocessor with its own resources, including fuses, SRAM and crypto hardware, acting as a **Hardware Root of Trust**. It runs a firmware signed by AMD and constitutes the **TEE** for the AMD's **Trusted Platform Module (TPM)**, which plays a crucial role in remote attestation [19]. **AMD-PSP** exposes a well-documented API for number of services, including generation and management of per-machine memory encryption keys, attestation of measured guest launch and migration support, among other commands.

2.5.2 AMD Secure Encrypted Virtualization

AMD Secure Encrypted Virtualization (AMD SEV) integrates main memory encryption with the AMD's virtualization technology, i.e., **SVM**. Each guest is associated with a unique encryption key used to protect data confidentiality while stored in **DRAM** [16]. The encryption keys are managed by **AMD-PSP**, ensuring that even the hypervisor cannot access the plaintext data. This presents a fundamental shift from the traditional ring-based security model of the **CPU**, where higher-privileged rings had unrestricted access to everything in the lower-privileged rings.

On system boot or virtual machine launch, a fresh memory encryption key is generated by the **AMD-PSP**. The encryption status of each memory page is determined by the value of the C-bit (encryption bit) in the page table entry. This allows the guest to specify which pages are private (C-bit = 1) and which are shared with the hypervisor (C-bit = 0)¹ [20].

The private pages are encrypted using the dedicated AES-128 (or AES-256 from the fourth generation of AMD EPYC) encryption engine built into every **CPU** memory controller. The correct encryption key is selected based on the **address space identifier (ASID)** of the executing guest, and AES-XTX mode is used to prevent malicious duplication of pages. When processed unencrypted on the **CPU**, all guest code and data are tagged with **ASID** and can only be accessed by the owner.

Once the machine starts, the guest owner may request that the virtual machine provide an attestation report, proving that it is running on a real SEV-enabled physical machine and that the initial memory image matches the uploaded one. The report is generated and signed by the **AMD-PSP**, whose signing key is connected through a chain of keys and certificates to AMD's root key, which acts as the root of trust.

¹The instruction fetches, and page table walks are always treated as private, regardless of the C-bit value.

2.5.3 AMD SEV-Encrypted State

The AMD SEV-Encrypted State (AMD SEV-ES) extends the AMD SEV features by providing confidentiality and integrity for the content of all CPU registers when the guest is not running [17]. When `#VMEXIT` occurs, the guest register state is encrypted before saving with the same key as used for memory. On `VMRUN`, the CPU decrypts the saved state and checks its integrity before switching to the guest context. To accommodate the encrypted state, the VMCB is divided into two areas. The first area contains various control bits, including the intercept vectors, i.e., events and instructions intercepted by the hypervisor. The second area, called the Virtual Machine Save Area (VMSA), contains an encrypted guest state [20].

The challenge with protecting the register state is that some services provided by the hypervisor, e.g., device emulation, require knowledge of specific registers. To realize these services while leaving the guest in control, the `#VMEXITs` are divided into two groups: Automatic Exits (AE), which require no guest state sharing, and Non-Automatic Exits (NAE), where the guest can specify what to expose. The Automatic Exits follow the same procedure as in SVM with an update to atomically load and store the encrypted state on the world switch. The Non-Automatic Exits, on the contrary, requires a new mechanism.

When an NAE event occurs, the CPU generates a VMM Communication Exception (`#VC`) that the guest must handle. Based on the NAE type, the `#VC handler` decides what to share with the hypervisor. The state is exposed through a shared memory page with a Guest-Hypervisor Communication Block (GHCB) structure [21] After copying the relevant state, the new `VMGEXIT` (Virtual Machine General Exit) instruction is called, and the hypervisor resumes execution. The hypervisor performs the emulation and places the new state into the GHCB before calling the `VMRUN`.

AMD SEV-ES protects against data exfiltration through the unauthorized reading of guest state and against control flow attacks, including rollback attacks that were still possible even with AMD SEV enabled.

2.5.4 AMD SEV Secure Nested Paging

Building on the AMD SEV and AMD SEV-ES features that ensured data confidentiality protection, AMD SEV-SNP adds data integrity guarantees [18]. Therefore, AMD SEV-SNP is the first technology that adheres to the threat model of confidential computing. In addition to integrity protection, AMD SEV-SNP also provides new architectural flexibility in the form of multiple virtual machine privilege levels (VMPLs), new attestation and key derivation architectures, and a flexible migration policy.

The basic principle of [AMD SEV-SNP](#) integrity is that if a guest can read a private (encrypted) page of memory, it must always read the value it last wrote. If the memory was modified, an exception should occur, informing the guest that the memory cannot be read. This principle is enforced through the new hardware-assisted data structure called the [Reverse Map Table \(RMP\)](#) and the page validation process, where the guest must approve the mapping of a page before it can be used [18].

The [RMP](#) is a large in-memory data structure tracking the ownership of every memory page. Every 4kB memory page is associated with a 16-byte [RMP](#) entry that indicates who is allowed to write to that page. Specifically, the [RMP](#) entry contains the [guest physical address \(GPA\)](#), the [ASID](#) to which the page is assigned, and the Validated bit. The [RMP](#) is not directly writable by software. Instead, special instructions were added to allow the hypervisor to assign a page to guests or reclaim the page.

When [CPU](#) or [input-output memory management unit \(IOMMU\)](#) (the only two options on AMD) wants to access memory, it performs a nested page walk to translate the [GPA](#) to the [system physical address \(SPA\)](#). After the address translation, the [SPA](#) is used as an index into the [RMP](#) and two properties are checked: (1) does the [GPA](#) match the value stored in the [RMP](#) entry, and (2) is the issuing process [ASID](#) the same as the one stored in the [RMP](#) entry [22].

The [RMP](#), as described so far, ensures that every [SPA](#) may be mapped to a single [GPA](#). The inverse should also not be possible, and that is why there is a Validated bit in the [RMP](#). The Validated bit is automatically cleared by hardware when the new entry is created for the guest. To use the page, the guest must issue *PVALIDATE* instruction to validate the translation. To ensure that a single [GPA](#) is never mapped to multiple [SPA](#), it is the guest's responsibility to validate valid memory corresponding to the same [GPA](#) only once.

In addition to memory protection, [AMD SEV-SNP](#) supports several optional features, including the [VMPL](#), interrupt injection restrictions, and fine-grained guest policy, which can specify whether the association with the migration agent is allowed or whether migration is disallowed altogether [18].

The [virtual machine privilege level \(VMPL\)](#) allow guests to subdivide their address space into four privilege levels. [VMPL0](#) is the most privileged level, with unrestricted permission to all memory pages. Any level may grant permission to a less privileged level, but it can never grant more privileges than it currently has. [VMPLs](#) enable the possibility of creating a new management inside the guest itself. This layer was later standardized as the [Secure VM Service Module](#) [23].

2.5.5 AMD SEV-SNP suggested migration

The official AMD SEV-SNP ABI specification [24] suggests two approaches for AMD SEV-SNP-enabled guest migration: (1) Swapping-based and (2) Guest-assisted. Both options rely on a Migration Agent (MA) that itself is an AMD SEV-SNP CVM bound to the primary CVM during the launch process. Single migration agents provide migration support for multiple machines simultaneously, but each machine can be bound to a maximum of one MA. As the MA is part of the guest's TCB, the attestation report also contains information about the MA.

The swapping-based approach can be performed without guest assistance, and as the name suggests, the hypervisor swaps out all guest memory pages and associated metadata pages. When swapped to disk, the pages are encrypted with the Offline Encryption Key (OEK). After all pages are swapped out, the machine is stopped. Migration Agent (MA) requests the guest's context page from the AMD-PSP and sends it to the target MA. This MA then starts the machine from the received context and begins swapping in the pages. The communication between the source and target MA is not specified.

The guest-assisted migration is based on Initial Migration Image software running in the guest context. This should result in faster migration, but no details are provided. From a conversation with Thomas Lendacky from AMD, it appears that AMD no longer plans to go in the direction of a migration agent, instead they will follow the SVSM-based approach.

2.5.6 Secure VM Service Module

The Secure VM Service Module (SVSM) is a component that exploits the VMPLs introduced as part of AMD SEV-SNP to help manage the guest lifecycle and provide secure in-guest services to the rest of the virtual machine, e.g., virtual Trusted Platform Module (vTPM), live migration support [23]. Figure 2.5 illustrates the component stack associated with using SVSM, highlighting the various elements operating within the CVM.

Even though the module was initially specified only for AMD SEV-SNP guests, there is a community-driven idea for a single management layer for all major confidential platforms: AMD SEV-SNP, Intel TDx and Arm CCA. Intel TDx support is not directly mentioned in the project description, but can be deduced from the source code and meeting notes, particularly from those dated 2024-05-08 and 2024-12-11 [25].

The SVSM should run on VMPL0, with the guest OS kernel running at a lower privileged level to isolate it from the services provided by the SVSM. The SVSM should provide functionalities that are usually performed by the hypervisor, such as access to the TPM. It is also anticipated that the SVSM will be used to support

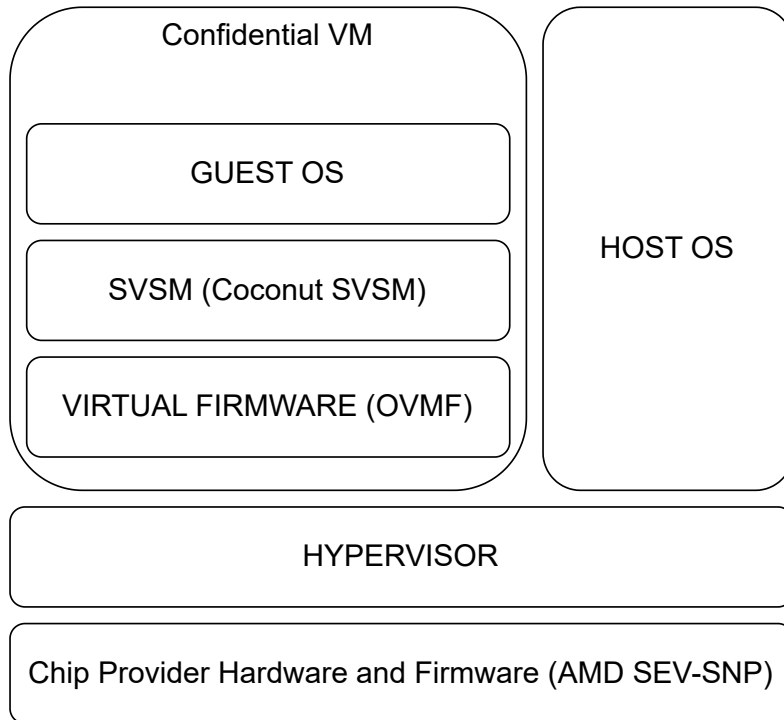


Figure 2.5: Components when running [AMD SEV-SNP](#) with [SVSM](#). Virtual firmware and [SVSM](#) run in [VMPL0](#), while the guest [OS](#) runs in [VMPL2](#).

fast live migration of machines.

The [SVSM](#) module specification [23] outlines the discovery process of the module by the virtual machine and the communication protocol between the virtual machine and the module. However, it lacks a detailed description of how the hypervisor should interact with the [SVSM](#) module. This communication is crucial for the hypervisor to initiate a migration of the guest or to prepare the guest for an incoming migration.

2.6 OVMF and IGVM

In contrast to standard (non-confidential) virtual machines, confidential guests necessitate the explicit provision of UEFI/BIOS firmware, such as [Open Virtual Machine Firmware \(OVMF\)](#), and require support for remote attestation. To facilitate the deployment of confidential virtual machines, a new file format has been developed [26].

The [Independent Guest Virtual Machine \(IGVM\)](#) file format is designed to encapsulate all the information necessary to run a virtual machine on any virtualization stack, supporting various isolation technologies such as [AMD SEV-SNP](#) and [Intel TDX](#), as well as untrusted guests. A specification with a reference implementation was developed by Microsoft and published as a Rust crate [26].

[OVMF](#)² is a format that provides UEFI support for virtual machines. The inclusion of firmware within the host is essential because each layer must be attested and therefore cannot be trusted to be provided by the cloud provider. [OVMF](#) is a widely used format and is considered a standard in cloud environments.

²In Intel documentation called [Trusted Domain Virtual Firmware \(TDVF\)](#)

3. Related work

This chapter first presents the mechanisms of live migration used by QEMU/KVM that are important for our implementation. Then, the existing approaches to the live migration of trusted execution environments are examined. The main concepts from various technologies are highlighted with emphasis on similarities and differences with the live migration of AMD SEV-SNP guests. This should provide the reader with a broader perspective on the topic and explain why existing implementations cannot be easily modified. Given the absence of publicly disclosed implementations for live migration of AMD SEV and AMD SEV-SNP technologies, the discussion consolidates concepts and proposals shared during conferences and on mailing lists.

3.1 Live migration in QEMU/KVM

Our solution builds on existing QEMU migration framework, so it is essential to understand how QEMU and KVM handle live migration for non-confidential machines. At its core, QEMU migration framework is highly modular: each device (RAM is also considered a device) registers a set of callbacks that run at key points in the migration process. Simple devices implement a save state hook on the source and a corresponding load state handler on the destination, while large-state devices add an iterative update function that is invoked repeatedly to send incremental state changes, and a finalizer to flush any remaining data. During migration, QEMU walks through all registered devices and invokes these callbacks in turn.

Modern QEMU has evolved toward multi-channel, multi-threaded migration. As virtual machines grow in both virtual CPU (vCPU) count and RAM size, with workloads exceeding 100 vCPUs and a 1TB of RAM memory, the ability to utilize several parallel data streams becomes crucial. Multiple channels increase the throughput and reduce the number of memory pages dirtied between iterations, which in turn brings the push-phase to its convergence watermark more quickly, shortening overall migration time.

Non-confidential dirty page tracking

In non-confidential environments, QEMU relies on KVM's dirty-page tracking mechanisms. All approaches build on the same underlying logic of marking pages as read-only, which triggers page-fault handling when a write operation is attempted. When a write exception occurs, the exception handler catches it, makes the page writable, and marks it as dirty.

The dirty-page tracking is enabled in the kernel through a control call¹ and retrieving a bitmap² indicating modified pages on each iteration. QEMU then re-transfers only those pages marked dirty. KVM offers two implementations for this: the traditional bitmap-based approach and the newer dirty ring interface [27], designed for high performance in large-memory environments. Since RAM typically accounts for most of the migration traffic, these optimizations are necessary to achieve acceptable live migration times for large virtual machines.

The main difference between the two approaches lies in their handling of the dirty log. In the bitmap-based approach, userspace periodically queries the KVM to retrieve the dirty log. Upon request, the KVM copies the entire bitmap to userspace, resets the page protections, and clears the dirty log.

In contrast, the dirty ring interface takes a different approach. It utilizes per-cpu circular buffers to store page frame numbers and a shared data structure between kernel and userspace to reduce the amount of copied data. This interface provides operations to add pages to the dirty ring when they are modified, and to scan or flush the dirty ring to process the tracked modifications.

3.2 Live migration of TEEs

The challenge of securely migrating encrypted memory pages has already been present in Intel Software Guard Extensions (Intel SGX) technology, in which each enclave, i.e., a protected area of execution that contains sensitive data and code, contains secrets that cannot be moved between different processors [15]. Because Intel SGX enclave secrets are tied to a specific platform, a new solution for live migration had to be found. A similar challenge involving encrypted memory pages that cannot be migrated using a supervisor also existed in AMD SEV technology.

Both Intel and AMD have advanced their technologies and introduced TEEs at the virtual machine level, Intel TDx and AMD SEV-SNP, respectively. Both technologies address the attacker model as specified in the thesis, but they have architectural differences.

On AMD platforms, security features are anchored in the AMD-PSP, which serves as a HRoT. The AMD-PSP is responsible for generating and safeguarding virtual machine encryption keys and for maintaining the Reverse Map Table, ensuring memory confidentiality and integrity across guest pages. By allowing multiple virtual machine privilege level, AMD enables the instantiation of a SVSM that executes with elevated privileges yet remains isolated from both the hypervisor and the guest.

¹Dirty page tracking is requested via `KVM_SET_USER_MEMORY_REGION` with the `KVM_MEM_LOG_DIRTY_PAGES` flag set.

²The dirty log is obtained using the control call `KVM_GET_DIRTY_LOG`.

Intel’s approach with **TDx**, by contrast, introduces two novel components: a privileged **CPU** operation mode known as **Secure Arbitration Mode (SEAM)** and a dedicated piece of trusted software called the **TDx** module. **SEAM** itself is partitioned into **VMX SEAM** root mode—where the **TDx** module resides—and **VMX SEAM** non-root mode—where each **TDx**, or secure virtual machine, executes. Within this framework, the **TDx** module running in root mode orchestrates key management, integrity checks, and migration policies on behalf of all **TDxs** operating below it [28].

The architectural divergence between **Intel TDx** and **AMD SEV-SNP** thus centers on where and how the root of trust is instantiated and managed. **Intel TDx** builds upon **Intel SGX**’s enclave concepts but relocates key functions into a privileged **CPU** mode and dedicated trusted software. **AMD SEV-SNP**, by contrast, relies on microcode extensions within the **AMD-PSP** rather than an entirely new **CPU** mode, consolidating both key management and migration policies in its secure processor.

3.2.1 Live migration of **Intel SGX**

In the context of **Intel SGX**, the enclave represents the **TEE**. The main challenge for migrating **Intel SGX** enclaves is private memory pages, which are encrypted using a hardware-based, non-migratable key. The proposed design is based on *Control Thread* running inside the enclave. When a signal is received, all other threads are paused, and the whole enclave state, except for a few pages containing enclave metadata, is re-encrypted with the migration key and placed in a shared memory, where the hypervisor can access it [29].

The transferred state is confidential and integrity-protected. The source machine and destination machine authenticate themselves through remote attestation, and only a single migration target is possible to prevent the fork attack. The migration encryption key is transferred at the end of the migration, and once sent, the source machine cannot be resumed. This mechanism is deployed to ensure that a single instance is running at all times. To prevent rollback attacks, all snapshots require the guest owner’s involvement and are logged by the enclave [15].

The high-level idea of where the guest should cooperate with live migration is the same for **Intel SGX** and **AMD SEV-SNP**. But the threat model is quite different as the **Intel SGX** does not trust the guest **OS**, while the confidential **VM** in **AMD SEV-SNP** case does not trust the hypervisor, but the guest **OS** (and **SVSM** if present) is in **TCB**. The size of memory is usually far smaller for the **Intel SGX** enclaves as compared to **AMD SEV-SNP** guests, and therefore, the whole enclave can be stopped and then transferred, which is not aligned with the definition of the live migration as viewed from the perspective of this thesis.

3.2.2 Intel TDx Migration guide

The TDx guide [30] proposes migration using Migration Agents (MAs). Each physical machine has a Migration TD that is running only the MA code. Once the migration is initiated, the source TDx, i.e., the context of the confidential virtual machine in Intel terminology, connects to a Migration TD that was specified during the machine launch. The Migration TD connects with the destination's Migration TD, and both check whether the other platform complies with the migration policy specified by the guest. If the migration policy check passes, then the Migration TDs negotiate an encryption key for the transport. The machines manage the rest of the migration, and a Migration TD is not required on either side.

The role of migration agents under Intel TDx is limited to the initial setup, whereas AMD's guest-assisted migration is also specified to assist during the migration process.

Intel provides the reference implementation of Migration TD in Rust [31]. However, the kernel virtualization module part is still a work in progress and proves to be a complex piece of software [32].

3.2.3 Live migration of AMD SEV-SNP

There are no publicly available solutions for live migration of confidential guests running under the AMD SEV-SNP. The most widely used userspace hypervisor, QEMU, explicitly states that the migration of confidential guests under AMD SEV (collectively referring to all AMD SEV, AMD SEV-ES and AMD SEV-SNP technologies) is not supported [33].

On the QEMU mailing list, IBM proposes an RFC [34] to add support for AMD SEV guest migration. The RFC builds on another RFC [35] that implements a migration helper in OVMF. The migration helper is implemented as a mirror VM, sharing the memory and encryption context (ASID) with the main VM. Unfortunately, the patchset is not relevant for AMD SEV-SNP migration, as private memory is managed differently under AMD SEV-SNP and AMD SEV, and the *guest-memfd* support was introduced for the confidential guests [36]. Another obvious difference is that the migration helper should be part of the SVSM, not implemented in the OVMF.

Another design, called PrometheusMigrate, was proposed for the live migration of AMD SEV-enabled guests [37]. Even though the source code is not published, the design description is detailed enough to understand the architecture. The migration helper runs on an extra vCPU inside the kernel OS and communicates with the hypervisor through two shared regions: (1) Transport buffer and (2) Message Queue as a communication channel between the hypervisor and migration helper.

While PrometheusMigrate is closer in design to live migration of [AMD SEV-SNP](#) guests than the previously introduced approaches, as it transfers pages to the target machine while the source machine is still running, the migration agent is again implemented as part of the guest firmware, i.e. [OVMF](#). For [AMD SEV-SNP](#) guests, the migration agent should be part of the [SVSM](#) module and the entire trusted guest is deployed using the [IGVM](#) file. This requires a re-evaluation of the design.

AMD explores the topic of live migration internally and presents pieces of the current state at the [KVM](#) Forums and Linux Plumbers Conferences. In 2021, AMD presented a status of Live Migration of Confidential Guests [38]. During this event, the [AMD SEV](#) was mainly discussed, but two main approaches to live migration are proposed: (1) Slow migration using the [AMD-PSP](#) and (2) Fast guest-assisted migration. The fast migration refers to the [IBM RFC](#) sets mentioned above. During the Linux Plumbers Conference in 2023, the focus shifted to [AMD SEV-SNP](#) guests [39]. While a high-level overview of how AMD envisions the Live Migration of [AMD SEV-SNP](#) guests with [SVSM](#) support was presented, the main discussion centred on the common APIs that all confidential architectures require. The latest public status update is from the last [KVM](#) Forum 2024 [40]. Two approaches for the [AMD SEV-SNP](#) guests with a [SVSM](#) live migration are proposed: (1) mirror [VM](#) and (2) shadow [vCPU](#).

In the mirror [VM](#) approach, a new machine is created (a separate [KVM](#) object), where [VMPL0](#) runs the [SVSM](#) Migration Helper. The hypervisor handles dirty page tracking, while the [SVSM](#) performs page packaging. Page packaging refers to adding metadata, hashing (for integrity protection) and encrypting the pages for transport. The first mirror [vCPU](#) is started when the migration is initiated. This is not currently possible, as *guest_memfd* used by [KVM](#) to map the private memory pages does not support sharing address space between the main and mirror [VM](#). There exists a patchset adding the required support [41]. The second approach uses a shadow [vCPU](#). The challenge here is that the shadow [vCPU](#) requires a non-overlapping [Advanced Programmable Interrupt Controller \(APIC\)](#) ID.

Both proposed solutions are viable, and the final version would be a compromise between the developers of the [KVM](#) module and [QEMU](#). The design of the migration helper running inside the machine is independent of the solution, and the main tasks, such as [RAM](#) migration, can be solved independently of the chosen solution. In this work, a design based on the ideas of [Intel SGX](#), [Intel TDX](#), PrometheusMigrate, and a general overview presented at the Linux Plumbers Conference in 2023 is used to determine the next steps and challenges in the field of live migration of confidential guests.

4. Methodology

This chapter outlines the research process and data collection methodology used to design, implement and evaluate live migration for confidential virtual machines. The necessary software is introduced.

4.1 Research process

The research process employed in this thesis was designed to be flexible and adaptable, allowing for changes in scope as needed. This approach was necessary due to the difficulty in estimating the complexity of individual steps beforehand. The chosen research process provides a suitable framework for future researchers to follow, as it effectively accommodates the dynamic nature of the research goals.

1. **State-of-the-art analysis:** Review relevant mailing lists, documentation, and source code to assess the existing functionalities and limitations related to the migration of confidential guests. This analysis helps to understand the current status of the project.
2. **Development of initial design:** Outline the overall architecture for migrating confidential guests while ensuring data integrity and security. This phase involves identifying the key components involved in the migration process and understanding the role of [AMD SEV-SNP](#) in safeguarding sensitive data.
3. **Identification of missing components:** Determine the specific functionalities or components that are necessary for successful migration. This includes documenting these gaps, their potential impact on the migration process, and their alignment with the scope of the thesis. Additionally, criteria for evaluating each component should be established.
4. **Research on implementation strategies:** Investigate various methods for implementing the identified components. It is essential to analyze the advantages and disadvantages of each approach in terms of performance, security, and complexity.
5. **Selection of the most suitable solution:** Evaluate the researched strategies against the established criteria. Choose the solution that best balances feasibility and effectiveness.

6. **Implementation of the chosen solution:** Develop a proof of concept based on the selected implementation strategy. Ensure thorough documentation of the implementation process, including any challenges encountered and the methods used to address them.
7. **Discussion of implementation results:** Analyze the outcomes of the implementation, focusing on both successes and failures. If the implementation is successful, outline the next steps for addressing additional missing components. Conversely, if the solution proves infeasible, document the reasons and consider alternative approaches or adjustments to the project timeline.
8. **Evaluation of our implementation:** Conduct a comprehensive assessment of the final implemented solution, focusing on its performance, security, and overall effectiveness. Providing recommendations for future work.

4.2 Data collection

This section outlines the process of collecting migration statistics to understand the cost of live migration for confidential virtual machines. The collected data provides information about [RAM](#) page migration, which is often the largest resource to migrate, and thus provides insight into the total cost of migrating confidential machines running under [SVSM](#).

The guest was subjected to two different types of load. In the first test, the amount of written memory was controlled directly and specified in megabytes of written memory per second. The minimum allowable downtime during which the machine successfully migrated was measured. In this test, we have direct control over the amount of dirty memory, and therefore other possible influencing factors are not relevant. In the second test, a specified number of connections to the [nginx](#) server running in the confidential virtual machine were created. This load was chosen to represent a situation closer to reality.

4.2.1 System setup

The launch script for virtual machine running the Coconut [SVSM](#) was modified to enable connections to the machine through [QEMU's Machine Protocol \(QMP\)](#) and [Secure Shell \(SSH\)](#). Additionally, modifications were made to allow the host to access the web server running inside the guest. A separate script was developed to run a non-confidential machine with an identical configuration.

Both confidential and non-confidential guests ran under the same patched [QEMU 9.1.50](#) to prevent any unforeseen influencing factors. The guests utilized

a modified 6.11.0+ Linux kernel with patches supporting confidential computing and custom changes for dirty page tracking through the *guest_memfd* interface.

The most significant machine configuration for our tests was: 8 GB of RAM memory, three vCPU usable by the guest, and one vCPU reserved exclusively for SVSM. We chose Void Linux as our distribution because of its small size and low performance overhead, allowing us to control the system load through additional programs.

4.2.2 Benchmark setup

All benchmarks followed the same setup. After booting the virtual machine, a connection was established via SSH, and the stressing workload was launched. In the case of the Nginx, the benchmarking process was run on the host computer. The maximum downtime was updated via QMP. The outgoing migration was initiated via QMP with a command equivalent to `migrate 'exec: cat > /dev/null'`. This approach allowed the migration stream to be written to a local file rather than being sent to another host, decoupling the sending and receiving implementations and enabling independent benchmarking of each side. To eliminate any factors related to file writing, the migration stream was redirected to `/dev/null`. Upon completion of the migration, statistics collected internally by the QEMU program were retrieved via QMP.

We proceeded according to the following experimental setup with the same setup was used for both confidential and non-confidential virtual machines. There were two main variables: load size and maximum downtime. For each load, we tried to find the lowest maximum downtime at which the machine would migrate reliably, while recording all other statistical data.

First, we determined the downtime required to migrate the machine without additional load. The decision tree illustrating the evaluation is shown in Figure 4.1 and described below.

A maximum of 50 migrations were performed for each workload. The maximum total migration time for a single migration was 10 minutes. This time was chosen to be relatively short in order to simulate situations where it is necessary to migrate a machine quickly and to shorten the duration of the experiment. If the migration was successful, another migration with the same downtime was performed until 5 consecutive migrations were successful. The current downtime was then set as the best downtime so far and subsequently decreased. If the migration was not completed on time, the maximum downtime was increased. If the migration was not completed on time and the best downtime had already been set, the best downtime was reported and next workload was tested. In rare cases where the migration was interrupted due to machine failure, the migration was rerun with the same maximum downtime.

In order to effectively find the best downtime, workloads were tested from the least memory-intensive to the most memory-intensive. The initial maximum downtime for the first workload is set to the time recorded for an unloaded machine. For each subsequent workload, the initial maximum downtime was set to the best downtime recorded for the previous workload.

4.2.3 Workload: stress-ng

In the first experiment, the system load was controlled with the *stress-ng* [42] tool, which provides an efficient way to stress the system. This tool allows us to focus on specific features, in our case, the rate at which RAM memory pages are written to.

The workload was started with the command `stress-ng --vm 2 --vm-bytes 10M --vm-method write64 --timeout 10m`. The `--vm` option specifies the number of processes started, `--vm-bytes` sets the total amount of virtual memory to be written to, `--vm-method` is chosen to just write to all the memory and `timeout` determines the duration of the stress test. System stress was controlled by changing the total amount of memory used.

4.2.4 Workload: Nginx

The second workload utilized the latest version of Nginx, 1.28.0, serving the default welcome page without any modifications. The host machine employed the *wrk* [43] benchmarking tool with its default settings: 2 threads and a duration of 10 minutes. The number of open connections was varied between 10 and 5000. Opening more than 7.000 connections often resulted in a `Too many open files` error, so we did not perform the test for multiple connections. By running the benchmark on the host machine, the overhead of the benchmarking tool did not impact the results, and benchmark statistics could be collected for successfully completed migrations.

Since we focused only on RAM migration and full machine migration does not work, the benchmark time was shorter if the machine was successfully migrated. The host machine performed one test at a time and, with 32 cores and 64 GB of RAM, did not represent a bottleneck.

4.3 Software

The virtualization stack we are using includes several key components: KVM, QEMU, and Coconut SVSM. The selection of the SVSM and a high-level overview of Coconut SVSM is provided below.

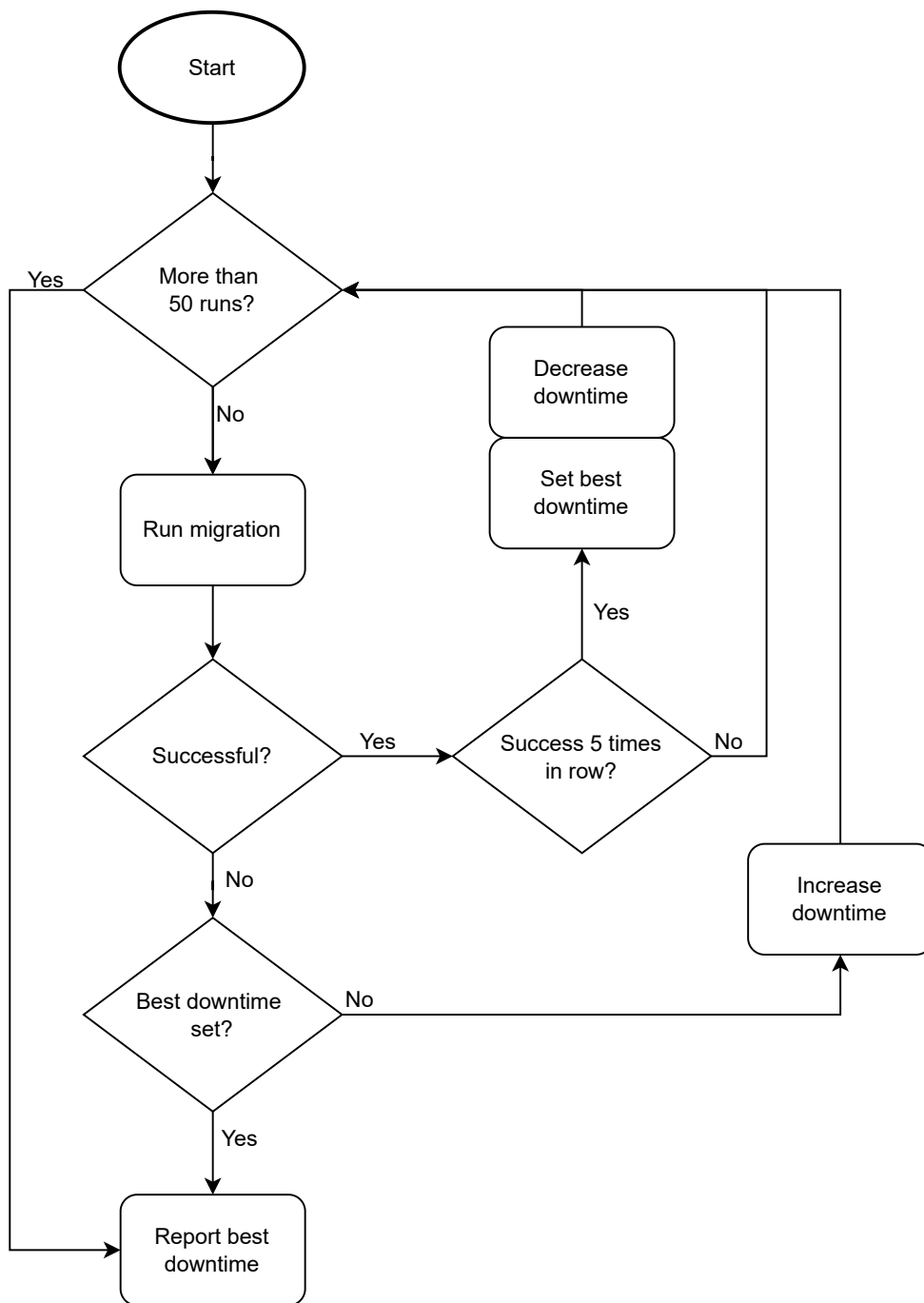


Figure 4.1: Experimental setup for a single workload.

KVM enables the Linux kernel to function as a hypervisor, leveraging the processor’s hardware virtualization capabilities. This enables virtual machines to operate at speeds comparable to native performance across a broad range of workloads. Notably, **KVM** can be utilized independently of **QEMU** and consists of a Linux kernel module—now integrated into the mainline kernel—that switches the processor into a guest world.

QEMU, on the other hand, operates in user space and serves as a virtual machine monitor, providing hardware emulation and a low-level interface for the virtual machine. Each **QEMU** process represents a virtual machine in its own right; it can be terminated by sending a signal to the process, and its resource consumption can be monitored using tools like `top`.

In order for confidential **RAM** migration to work, changes were made to the source code of all **SVSM**, **QEMU**, and **KVM**.

4.3.1 Choice of **SVSM**

The **Secure VM Service Module (SVSM)** specification [23] was published and therefore allows for multiple implementations of the specification. We found two public implementations of the module (1) **Verismo** [44] and (2) **Coconut-SVSM** [45].

Verismo is an open-source **SVSM** module for AMD **CVMs**, developed by Microsoft and available on GitHub [46]. Implemented in Rust, **Verismo** has undergone formal verification with **Verus** [47]. It targets the most recent AMD confidential computing architecture, **AMD SEV-SNP**, and integrates with a Windows-based hypervisor. However, repository activity suggests that it is not actively developed.

Coconut-SVSM, originally developed internally at SUSE and open-sourced in March 2023 [48], evolved from the now-discontinued Linux **SVSM** effort [49][50]. Its development community includes researchers from AMD, Red Hat, Google, and Microsoft, and it recently joined the Confidential Consortium [51]. **Coconut-SVSM** shows strong community support and aims to provide broad support for multiple hypervisors (**KVM** on Linux and **Hyper-V** on Windows) and various confidential computing technologies (**AMD SEV-SNP** and **Intel TDX**).

Currently, **Coconut-SVSM** appears to be the most advanced **SVSM** module available, which is why we decided to examine and advance its live migration support.

4.3.2 **Coconut-SVSM** overview

Coconut-SVSM is developing rapidly, with weekly meetings and number of commits every week. Documentation is practically non-existent, and due to the rapid changes, it is impossible to capture the latest state of the project. Here we describe the parts of its architecture that are important for the rest of this work.

On AMD SEV-SNP, the SVSM kernel code runs at the VMPL0 level (the highest privilege level), the SVSM user space at the VMPL1 level, and the host kernel at the VMPL2 level. Services provided to confidential virtual machine should be run as user-mode processes by default and only implemented in the SVSM kernel when necessary [52]. Since the SVSM user space is not yet implemented, we wrote all the code in the SVSM kernel. Both the hypervisor and the host kernel must run on a patched version of Linux [53] or hyperV. During prototyping, we used the Linux version. QEMU support for IGVM files has already been merged and will be part of the QEMU 10.1 release [54]. AMD SEV-SNP support has been part of QEMU since version 9.1 [55]. Therefore, the patched version [56] we used should not be needed in the near future.

The SVSM kernel boots and initializes before it begins listening for requests from the above-running guest kernel. SVSM acts as a communication intermediary responsible for processing guest operating system requests as defined in the SVSM specification [23], providing the hypervisor with relevant information on the non-automatic #VMEXITS, and communicating with the hypervisor via the GHCB [21]. The GHCB specification explicitly states that it does not define the mechanism by which the hypervisor communicates with the SVSM, as it is assumed that these details are specific to the host environment [23]. SVSM runs only if an action is requested either by the guest or by AMD Platform Security Processor (AMD-PSP).

With the virtual Trusted Platform Module (vTPM) disabled, the host currently issues only one call, *SVSM_CORE_PVALIDATE*, which requests validation of the translation in the RMP table. Based on this request, SVSM updates the RMP permissions and validates the translation. Since SVSM is not involved in memory access, page validation is the only action that can be used to track the status of guest pages.

Guest pages are stored in a lock-protected global structure called *MEMORY_MAP*¹ and can be read and written using dedicated functions.²

¹<https://github.com/coconut-svsm/svsm/blob/7e0528bb783f59645782d68059b168ef5241a06d/kernel/src/mm/memory.rs#L22>

²<https://github.com/coconut-svsm/svsm/blob/7e0528bb783f59645782d68059b168ef5241a06d/kernel/src/mm/guestmem.rs>

5. Design and Implementation

This chapter answers research questions R1 and R2 by presenting a design for live migration based on an analysis of the current state of the art and examination of the code bases of relevant open-source programs. The design aims to address the challenges of live migration, and the following sections discuss the development of tasks towards achieving this goal.

5.1 Design

We propose a design for live migration of **confidential virtual machine (CVM)**, addressing research question R1. Building on the existing **QEMU** migration framework, incorporating feedback from the community on our draft pull request,¹ avoiding the need for new commands while ensuring scalability to additional **vCPUs**.

The design process, shown in Figure 5.1, involves four key actors: the trusted **SVSM** instance at the source, the trusted **SVSM** instance at the destination, and the untrusted hypervisors at both ends. On the source side, the hypervisor comprises **QEMU** in user-space and **KVM** in kernel space, whereas on the destination side only user-space component plays an active role in the migration, therefore the kernel space part is omitted from the diagram.

The migration process starts with the **CVM** running on the source. On the target side, the **CVM** is launched from the same **IGVM** image as the source **VM**, with all **vCPUs** stopped except for the **Migration Handler**'s **vCPUs**. The launch of the target machine must be modified to allow the **SVSM** and its migration handler to start even before the migration is started. Required changes likely affect both **QEMU** and **OVMF** firmware, though we have not implemented this step.

Once the destination machine is ready for incoming migration, the source **QEMU** issues a *migrate* command to begin the outgoing migration. Source **QEMU** then notifies its **SVSM**, which launches the migration handler to oversee the transfer of encrypted state, e.g., **RAM** pages. This migration handler runs on additional **vCPUs** that are added once migration has started.

Before proceeding, the source and target environments must perform mutual attestation to verify the integrity of their trusted execution environments. To ensure the trustworthiness of this process, the **SVSM** instances must establish a shared secret, from which a secret for attestation and then a secret for encrypting the migrated state is derived. Attestation failure will cause the migration to be aborted. On AMD platforms, guest owners can further enhance security by completely disabling virtual machine migration, although this measure may reduce

¹<https://github.com/coconut-svsm/svsm/pull/745>

system reliability and availability. The continuity of the attestation chain is not in scope of this thesis.

After successful attestation, the push phase is entered. Source **QEMU** asks its migration handler to package each private **RAM** page, i.e., assigning metadata, encrypting with the shared secret, applying integrity protection, and writing the result to a shared page so that **QEMU** can read it. Non-confidential memory and device state continue to use **QEMU**'s existing migration mechanisms.

Live migration proceeds in iterations: **KVM**'s dirty-page tracking identifies guest **RAM** pages modified since the previous round, and only those pages are re-packaged and sent. Each message carries a monotonically increasing tag and integrity checksum is computed over all pages sent, assuring the destination migration handler that no duplicates or damage occur. Upon arrival, **QEMU** on the target passes encrypted pages to the **SVSM**, which decrypts and writes them into guest memory.

Since **SVSM** is not involved in memory access, it cannot track dirty pages on its own. Therefore, we decided to use **KVM** to monitor dirty pages. This decision represents a potential attack vector. A detailed discussion of this decision is described below.

When the remaining dirty pages fall below a predefined watermark, the **SVSM** initiates a stop-and-copy phase. All guest **vCPUs** are halted and two-phase checkpoints ensure that the hypervisor cannot run guest **vCPUs** during this phase. Final iteration transfers remaining **RAM** pages and the machine's **VMSA** state using existing routines. The **SVSM** state at the destination should be updated so that **SVSM** services, such as **vTPM**, can be restored. Updating the **SVSM** is not covered in this work.

As the last step, the source **SVSM** issues an integrity report, e.g., the count of transferred pages and/or a composite hash of their addresses. Passing this check triggers the destination to start the **CVM**. Since the source has already stopped, no formal acknowledgement is needed.

Challenges we have not addressed in our design are post-copy live migration and management of swapped-out pages.

5.2 Starting the migration handler in **SVSM**

This and subsequent sections describe the tasks required for live migration of confidential machines.

Initially, the hypervisor must notify the **SVSM** that a migration has been requested. In a non-confidential case, the machine was not aware of the ongoing migration. This is no longer efficient, and guest cooperation is required, therefore, the guest, or **SVSM** in our case, must be notified.

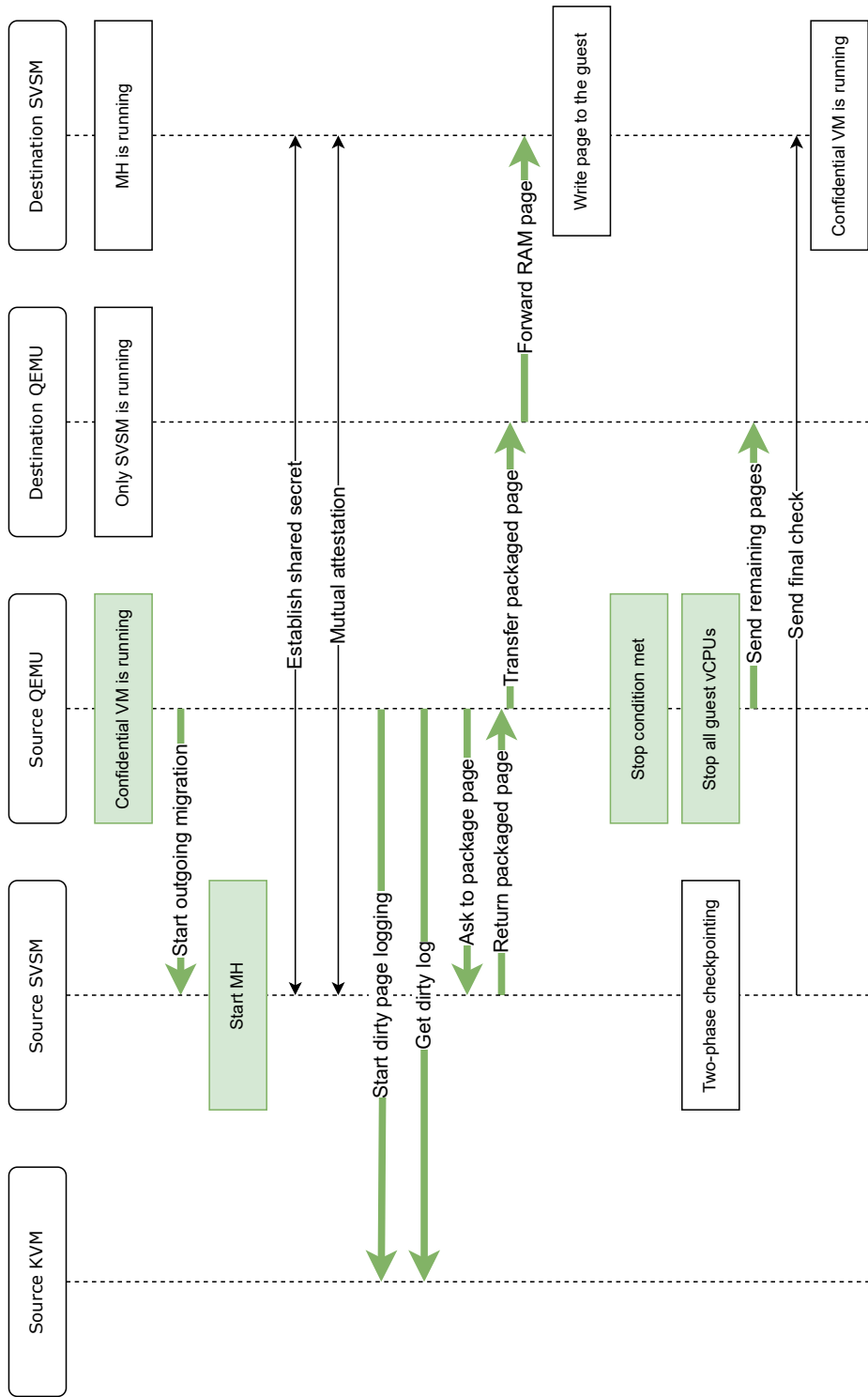


Figure 5.1: Design for live migration of confidential virtual machines. The parts highlighted in green have been successfully implemented.

There is no specified mechanism to invoke the **SVSM** from the hypervisor. We came up with two options, how this may be implemented: (1) hypervisor issues an interrupt that is directly handled by the **SVSM** (if such an interrupt exists), (2) hypervisor issues a newly-defined interrupt handled by the guest and when handling the interrupt, the guest performs a newly-defined request to the **SVSM**.

Once the **SVSM** is notified about the migration, the hypervisor should start a migration handler on a new **vCPU(s)**. This may be implemented as either a mirror **VM** or through shadow **vCPU** as outlined in Section 3.2.3.

Although the interrupt seems like the optimal solution to notify the **SVSM** about the migration, for proof of concept, we start the **SVSM** on an extra **vCPU** that is not usable by the guest **OS** and only busy waits for the migration to start. Our solution has a scalability limitation as additional **vCPUs** cannot be added dynamically. More **vCPUs** should not be a problem, because the pages to be packed can be distributed among the **vCPUs** and no synchronization is required.

5.3 **QEMU** and migration handler communication

During the migration, there needs to be a mechanism by which the **QEMU** can ask the migration handler to package certain pages and for the migration handler to share the pages ready to be transferred.

There are two possibilities: (1) interrupt and (2) memory variable. In both cases, a shared page is used to transfer data. In interrupt-based, **QEMU** issues an interrupt to inform the migration handler that the next page packaging was requested, and a new exit reason is defined for the migration handler to inform **QEMU** that the next page is ready. In a memory variable option, **MH** periodically checks a certain variable, and upon the change of that variable, the **MH** performs a task and updates that variable once the task is finished.

We designed our solution around the second option, as we expect the migration handler **vCPU** to be occupied during the migration, and the interrupts, therefore, seem excessive. We expect the migration handler performance or the network bandwidth to be the bottleneck.

Our implementation consists of a single shared page called **MigrationPage** that is used for data transfer and communication with **QEMU**. The migration page contains two registers and a buffer: a status register, a data register, and a data buffer. The status register is used to signal a change in status (e.g., migration starts, migration is complete). The data register is used to signal that a new page has been prepared in the data buffer by the provider or processed by the consumer. The roles of provider and consumer are switched between **SVSM** and **QEMU** on

the source and destination machines.

5.4 Source **SVSM** to destination **SVSM** communication

The source and destination **SVSM** need to communicate over an untrusted channel. When establishing a secure channel between the source and destination **SVSM** over an untrusted network, a crucial decision is whether to integrate the networking stack within the **SVSM** or leverage the hypervisor’s existing networking capabilities. Opting to rely on the hypervisor’s networking stack, specifically **QEMU** in this case, eliminates the need for additional code in the trusted domain, thus reducing unnecessary complexity.

Two primary options exist for realizing this channel: (1) establishing a new direct connection, such as a socket connection, between the source and destination **QEMU**, or (2) modifying **QEMU**’s migration functions to accommodate confidential machine migrations. The first approach offers the advantage of independence from the hypervisor, which is not considered trusted in our attacker model, but it also requires the **SVSM** to handle the entire migration process, introducing complicated logic that must be reimplemented within the guest or **SVSM**.

The second option involves utilizing **QEMU**’s established migration framework and registering custom functions for migrating confidential guest’s **RAM** pages. This approach allows for the seamless migration of disks and other peripherals without additional effort. However, a new challenge arises: the migration handler and **SVSM** must already be operational on the destination before **RAM** page migration commences. Otherwise the destination **QEMU** had no one to forward the packaged pages to.

Exploring the first option yielded feedback from the community indicating that introducing a new specialized command solely for migration of confidential guest would not be viable for **QEMU**.² As a result the second solution was pursued.

To implement the second solution, updating the **RAM** migration functions is relatively straightforward, involving the modification of function registrations within the *migration/ram.c* file in **QEMU**, specifically through assignments to the *savevm_ram_handlers* struct.

²Our pull request based on the first option: <https://github.com/coconut-svsm/svsm/pull/745>.

5.5 Tracking validated pages

In a confidential computing environment, every guest page must be validated before it can be used. The validation process involves a hypercall to the **SVSM**, which then issues a call to the **AMD-PSP** to mark the page as validated in the **RMP** table. Additionally, the page must be granted **RMP** access to be usable by the guest **OS**, as the **AMD-PSP** maintains an **RMP** table that tracks which **VMPL** levels can access which pages and with what permissions.

To efficiently manage validated pages, it is beneficial for the **SVSM** to track this information. Currently, this information is not stored in the Coconut-**SVSM**, but it is necessary to ensure that only validated pages are transferred during migration. There are several ways to obtain the validated pages: (1) querying the **AMD-PSP** for each page's validation status, (2) maintaining a structure like a bitmap within the **SVSM** to mark validated pages, or (3) having the guest **OS** maintain the list of validated pages.

Querying the **AMD-PSP** for the validation status of each page is a secure option as the **AMD-PSP** is trusted, but it has a performance impact due to limitations of the **AMD-PSP** chip. The command has also disappeared from the latest 1.58 revision of the **AMD SEV-SNP ABI** specification [24],³ so it is unclear whether it will be supported in the future. In addition, it may be necessary to support multiple protocols for different trusted architectures, which may be impractical or increase complexity.

Having the guest **OS** maintain the list of validated pages and providing an interface to extract this information could work but would require either a single solution across various operating systems or the **SVSM** to support multiple protocols. This approach introduces additional complexity and may not be scalable.

Therefore, the chosen approach is to implement a tracking structure within the **SVSM**, such as a bitmap, to mark validated pages. This approach allows for a common denominator across various guest **OS** and confidential platforms, at the same time facilitating extensibility if there would be a need to track other per-page information in the future, such as privileges or whether a page is shared or private. However, a challenge with this approach is that the size of the **SVSM** kernel memory is currently decided statically when the **IGVM** file is generated, which can limit the support for features like adding additional memory during runtime. Yet, support for memory ballooning is out of scope for now [58].

There is an ongoing discussion in Coconut-**SVSM** community about implementing a per-page tracking structure, potentially with some form of compression, such as storing information about page ranges instead of individual pages. This is based

³The command can be found in section 5.3.4 Additional Instructions for Managing x86 Pages in revision 1.57 from January 2025 and in the following article [57]. We were unable to find revision 1.57 online, but we have a local copy that we can provide upon request.

on the observation that pages close to each other often have similar statuses. The outcome of the discussion is that it makes sense to implement what is currently needed and expand it in the future. Changing the implementation of the tracking structure does not seem to be a major concerns [58].

Therefore a bitmap without compression was implemented for live migration purposes. Every time a page validation is requested from the **OVMF** or guest **OS**, the handler sets the corresponding bit in the bitmap of validated pages. The correctness of the tracking process can be verified by attempting to validate all pages marked in the bitmap and invalidating any other page, both of which should yield a *FAIL_UNCHANGED* status, indicating that the page was already in the state before the validation was issued. Even for a guest memory size of 8GB, the maximum allocation size of the **SVSM** kernel needed to be increased.

5.6 Dirty page tracking

A key aspect of live migration is tracking dirty pages to determine which pages need to be transferred during each iteration. Page tracking is traditionally performed by modifying the page table attributes. In the context of virtualized environments, two page tables are involved: one within the guest **OS**, translating guest virtual addresses to guest physical addresses, and a nested page table, translating guest physical addresses to system physical addresses. During page translation, the guest interacts directly with the hypervisor (**KVM**), which means that only the guest and the hypervisor kernel space can track dirty pages. However, when dealing with confidential guests, the hypervisor is not trusted therefore, the page tracking should be addressed with special care, and a verification mechanism should be implemented.

5.6.1 **QEMU** *guest_memfd* interface

In addition to the encryption of private memory, the new *guest_memfd* interface in **KVM** presents another technical challenge. This interface was introduced to prevent accidental modifications to guest memory and enhance security. A significant change was made, where guest private memory is no longer mapped in the hypervisor userspace [59][60][61].

Prior to the introduction of *guest_memfd*, the migration handler could read guest pages, even if they were encrypted, and forward them to the migration handler for packaging. However, with the new model, **QEMU** can no longer access the pages as they are not mapped in its address space. Furthermore, **KVM** does not support dirty page tracking on memory backed by *guest_memfd*.

One possible solution considered was to force **QEMU** to use a different method to back the confidential guest's memory, but this seemed like a step backward. Moreover, the amount of work required for this change did not seem to justify the effort. As a result, the current limitations of the *guest_memfd* interface pose a challenge to be addressed.

5.6.2 Possible solutions for confidential guests

We considered three options for tracking dirty pages in a confidential computing scenario. The first option involved removing write privileges for the guest privileged level from the **RMP** table for every validated page. This approach triggers the *handle_rmp_fault* handler in **KVM**. While this handler could potentially inform the **SVSM**, it would require frequent context switching between the hypervisor kernel space and the guest kernel space, which is expected to be an expensive operation. At the same time, this option depends on the hypervisor informing **SVSM**, so this solution was not explored further.

The second option is to perform page tracking inside **KVM**, then request a bitmap of dirty pages in **QEMU**, and then pass it to **SVSM**. **SVSM** can read the bitmap and package the relevant pages. Alternatively, **QEMU** can send **SVSM** only the memory addresses of the pages to be packaged. Currently, it is unclear to us how the bitmap map obtained from **KVM** maps to the *MEMORY_MAP* structure where guest memory pages are stored inside Coconut-**SVSM** and because of *guest_memfd*, **QEMU** cannot read the pages directly.

The third option is to track dirty pages in the guest's page table. This seems most suitable for a confidential computation scenario, because the guest is trusted and also participates in address translation. **QEMU** can then inform the guest kernel that another migration iteration has been initiated, the guest kernel sends the bitmap to the **SVSM**, and starts tracking dirty pages from the clean bitmap again. At the beginning of the iteration, the guest operating system (or **SVSM**) also provides the hypervisor with statistics on the number of dirty pages so that **QEMU** can still decide when to enter the stop-and-copy phase.

This option represents an obvious compromise in that the guest operating system must be enlightened and support confidential computing, which contradicts the idea of **SVSM**, which should handle all these services even for guests that do not support confidential computing.

We decided on the second option, even though tracking dirty pages is not officially supported for *guest_memfd* backed memory. Nevertheless, we managed to successfully track the pages by removing explicit blockers and updating the **KVM** source code in several places.

5.6.3 Security analysis

Tracking dirty pages with a hypervisor raises two potential security issues: a rollback attack and information extraction.

In the case of a rollback attack, if the hypervisor falsely reports that there are no dirty pages, even though there actually are some, it would be possible to restore the machine to its previous state, essentially turning back time. This type of attack is not specific to live migration, as it is essentially the same process as recording pages at a certain point in time and then restoring them in the future. One possible solution to mitigate this risk is to calculate a memory hash after stopping the machine or to compute a cumulative live hash for all transferred pages and then include the result in the final check.

The second issue is information extraction. Apart from physical side channels, which are out of scope for confidential virtual machines, the host may observe the memory access patterns of the guest, i.e., the time and pages accessed, since the nested page table is located in the host. This can be exploited through page fault handling, where an attacker could potentially infer information from the observed behaviour.

5.7 vCPU stopping

The live migration finishes with a downtime phase, where all CPUs are stopped, and the remaining state is transferred. To ensure a reliable execution of this phase, the SVSM must guarantee that the hypervisor cannot start any vCPUs at any time.

This requirement leads to the need for a two-phase checkpointing mechanism [29]. The idea is to have all guest threads enter a busy loop, effectively pausing their execution. Even if the hypervisor attempts to run any thread, no actual code will be executed, ensuring that the system remains in a consistent state.

The necessity of this mechanism can be illustrated by the example of a data consistency attack [29]. The hypervisor confirms that the vCPUs are stopped without actually doing so. As a result, the machine will continue to operate and the system state may be inconsistent. This inconsistency could allow a malicious hypervisor to exploit the situation, which could lead to a security breach.⁴

To achieve this, the SVSM needs to run a small piece of code that checks whether a certain variable is set or not. If set, it enters the busy loop, which is never exited as it marks the end of the migration process.

⁴The example is adaptation of example from the original paper [29].

However, a challenge arises in determining where to place this code, as there is no periodically executed code in the Coconut-SVSM. One possible solution is to issue an interrupt that should be handled by the SVSM, which would then check the variable. The question remains as to who should issue this interrupt: (1) a hypervisor that is not trusted, or (2) a guest operating system that would periodically send requests to the SVSM. Furthermore, the migration handler has no way of knowing when the last phase of the migration has started, as the hypervisor only sends RAM pages for packaging and does not provide information about state of migration. If the migration handler knows that the final phase has started, it can refuse to send the final check until all threads are busy waiting, and therefore it may be the hypervisor that issues the interruption without there being any potential security risk.

5.8 Save zero page optimization

The *save_zero_page* is an optimization in QEMU, which, instead of transferring the entire page, simply sends the information that the page contains only zero values, thus significantly saving bandwidth.

In our experimental setup, we emulate this behaviour by having QEMU query SVSM for the number of validated pages in the initial round. Subsequently, QEMU transfers this specified number of pages as normal pages, while the remaining pages are transferred as zero-filled pages.

In a real implementation, SVSM should also provide the addresses of all validated pages so that the correct pages are transferred. The implementation of *save_zero_page* optimization would not lead to any new information leaking into the hypervisor, because the hypervisor is able to track which pages are being written to using the same mechanism that is used to implement dirty page tracking, i.e., by modifying the flags of the nested page table.

Note that currently, the address translation between the address sent by QEMU cannot be mapped to addresses used in Coconut SVSM; therefore, for every request, a random validated page is read, packaged and returned.

5.9 Summary

This chapter details presented our design for confidential guests. Our design and implementation of live migration for confidential virtual machines builds on the existing QEMU migration framework.

The design guarantees that confidential memory pages are encrypted and protected for integrity before exiting the trusted domain, whereas non-confidential

state is migrated using established [QEMU](#) mechanisms. However, some aspects remain to be addressed, such as post-copy live migration and management of swapped-out pages. Furthermore, the implementation of live migration for [CVMs](#) requires careful consideration of security risks, including rollback attacks and information extraction. We propose two-phase checkpointing mechanism to ensure that the hypervisor cannot start any [vCPUs](#) during the downtime phase, maintaining the consistency of the system state.

Sections 5.2 to 5.7 address the research question R2, presenting the key components and their characteristics that are necessary for live migration of confidential guests. Specifically, we identified five key components that are missing today for live migration under [SVSM](#):

- [SVSM](#) migration start: A mechanism is required to initiate the migration process, which should be started by the hypervisor on a single or multiple shadow [vCPUs](#). We solved this using a single busy-waiting [vCPU](#) waiting for a signal that migration has started. The [vCPU](#) is made inaccessible by the guest through UEFI firmware changes.
- Hypervisor ([QEMU](#)) and [Migration Handler](#) communication: A specification of a channel between the hypervisor and [MH](#) to manage the migration process, including information about dirty pages and the transfer of the packaged pages.
- Validated pages tracking: Only validated pages should be transferred, so a mechanism is required to track these pages. We implemented this using a bitmap inside the [SVSM](#).
- Dirty page tracking: A mechanism is needed to track dirty pages. This is a technical challenge related to a newly introduced `guest_memfd` interface, which is used to map guest private pages. We updated the code that allows us to track dirty pages even for `guest_memfd`.
- Stopping the source machine: The [SVSM](#) should ensure that the source machine is stopped before allowing the destination machine to start. This was not addressed in our implementation.

6. Evaluation

This chapter answers the research question R3 by evaluating the performance of our proof-of-concept for confidential virtual machine migration, assessing its migration speed and comparing it to that of non-confidential machines.

6.1 Migration speed: stress-ng

The first experiment is conducted in a controlled environment, which enables us to explicitly specify the speed of writing to memory. The main goal of this experiment is to estimate the slowdown of migration speed for our solution. The migration speed is measured in pages per second.

The first experiment is done in controlled environment, that allows to explicitly specify the speed of writing in the memory. The main goal of this experiment is to estimate the slowdown of migration speed for our solution. The migration speed is measured in pages per second.

6.1.1 Evaluation setup

Virtual memory usage was tested for values of 10, 15 20, 30, 50, 75, 100, 200, 300, 400, 500 megabytes per second. For each workload, the shortest maximum downtime was determined using the methodology described in Section 4.2. For the confidential machine, the downtime step was set to 0.5 seconds for maximum downtimes under 15 seconds and to 1 second for longer downtimes. For non-confidential machines, the maximum was set to 0.1 seconds for all workloads.

6.1.2 Results

The maximum downtimes recorded for confidential and non-confidential machines are listed in Table 6.1 along with the recorded migration speed in pages per second. The values were calculated as the median of the five consecutive successful runs of a given workload for the specified maximum downtime. The number of five successful runs was chosen to provide sufficient confidence while completing the experiments within the limited time available, with access to the AMD SEV-SNP machine.

We can observe that the downtime required for reliable migration is 9 to 15 times longer for confidential machines, while the migration speed of confidential machines is 7.4 times slower than that of non-confidential machines.

In terms of migration speed, we found that for non-confidential machines, the variance is low and there are no significant changes across different workload sizes. In contrast, for confidential machines, there is a noticeable speedup for larger workloads. We do not know the cause of this acceleration.

Load (MB/s)	C down (ms)	C speed (pps)		NC down (ms)	NC speed (pps)		Slowdown
		Median	Mean \pm SD		Median	Mean \pm SD	
10	1,500	4,393	4,401 \pm 17	100	33,000	34,066 \pm 2,361	7.51
15	2,000	4,431	4,416 \pm 21	200	33,270	35,376 \pm 4,036	7.51
20	3,000	4,431	4,439 \pm 42	200	32,960	33,038 \pm 110	7.43
30	4,000	4,431	4,446 \pm 34	300	32,960	34,936 \pm 4,302	7.43
50	6,500	4,431	4,416 \pm 21	600	32,960	32,946 \pm 198	7.43
75	9,500	4,431	4,568 \pm 270	800	32,960	34,018 \pm 2,272	7.43
100	11,500	4,431	4,439 \pm 17	1,100	32,960	32,894 \pm 146	7.43
200	22,000	4,431	4,664 \pm 320	2,100	32,960	33,500 \pm 741	7.43
300	32,000	4,431	4,643 \pm 293	2,900	32,960	32,962 \pm 4	7.43
400	39,000	4,548	4,697 \pm 315	3,700	32,960	34,652 \pm 2,530	7.25
500	43,000	5,039	5,046 \pm 17	4,600	32,960	32,962 \pm 4	6.54
average	—	4,431	4,561 \pm 184	—	32,960	33,759 \pm 93	7.43

Table 6.1: Shortest reliable maximum downtime at different memory write speeds per second. The downtime for non-confidential (NC) and confidential (C) machines is given in milliseconds (ms) along with the migration speed in pages per second (pps). The slowdown is the ratio of non-confidential to confidential migration speed.

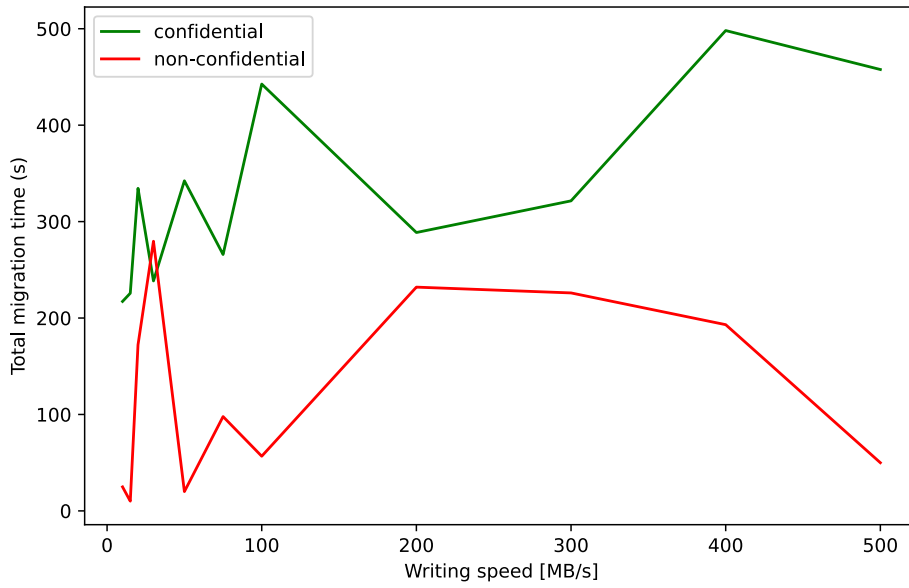
Figure 6.1 confirms that RAM migration is the dominant factor in the total live migration time for both confidential and non-confidential scenarios. The identical behavior between the two subfigures is clear. When the red/green line in subfigure (a) rises, the same pattern occurs in subfigure (b), and vice versa. This correlation demonstrates that the total number of migrated pages and the total migration time curves are closely linked in both cases. The total number of migrating pages is significantly higher for non-confidential machines, even when the total migration times are shorter, due to a much higher pages-per-second rate.

6.2 Real world scenario: Nginx & Wrk

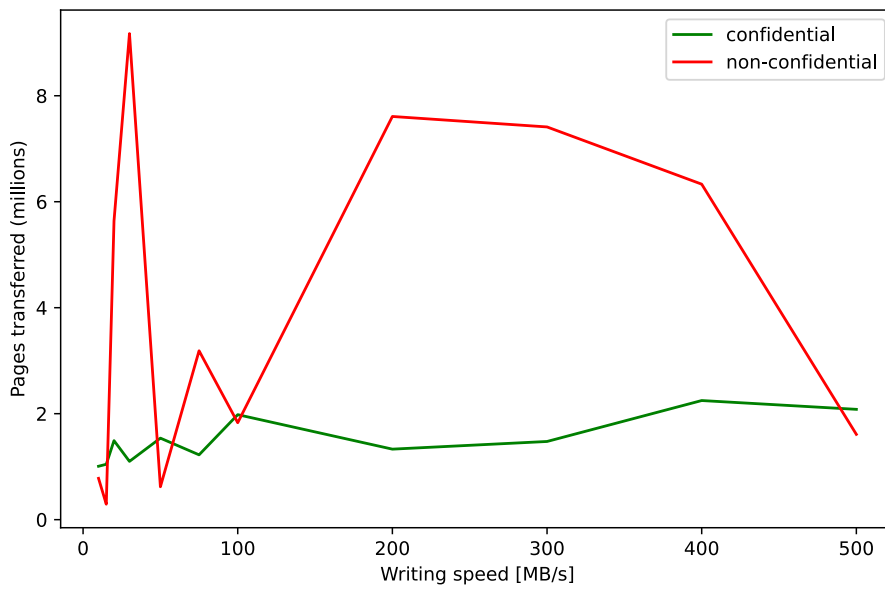
The Nginx & Wrk workload was performed to simulate a real-world scenario.

6.2.1 Experimental setup

Using the methodology outlined in Section 4.2, we determined the shortest maximum downtime for 10, 50, 300, 400, 600, 2000, 5000 connections being open at



(a) Non-confidential machine.



(b) Confidential machine.

Figure 6.1: The total number of pages transferred and the total migration time.

every single moment to the Nginx server serving a static website. The number of connections were controlled by a wrk tool [43]. Similarly, to the *stress-ng* the downtime step was set to 0.5 seconds for maximum downtimes under 15 seconds and to 1 second for longer downtimes in case of confidential machines and 0.1 seconds for non-confidential machines.

6.2.2 Results

The results, presented in Table 6.2, shows a similar migration speed slowdown factor 7.2 and low variance as in the first experiment.

On the other hand, the non-confidential machine consistently completed tasks with the default maximum downtime of 300 milliseconds, while the confidential machine experienced a significant increase in downtime, which was unexpected given that the RAM migration speed remained comparable to that observed in the *stress-ng* workload.

The migration speed and number of total transferred pages remains consistent with the results observed for the *stress-ng* scenario. On the other hand the recorded downtime shows a significantly different trend, specially for the non-confidential machine. The minimal allowed downtime is high directly from the low number of connections, for moderate number it is already at the highest point or around 16 seconds and this does not change for even large number of connections. The different results compared to the first *stress-ng* point to need for more distinct benchmarks.

6.3 Discussion

Our experiments show a substantial decrease, 7.4 and 7.2 times, in RAM migration speed for confidential machines compared to non-confidential machines. This performance difference can be attributed to two main factors. Firstly, QEMU's inherent support for multithreading enables multiple pages to be read and queued for transmission over the migration stream, but this capability is not fully utilized in the current implementation. Secondly, there is a necessary overhead associated with the interaction between QEMU and SVSM, where QEMU signals to SVSM that the next address has been sent, and SVSM must validate, encrypt, and signal back that the packaged page is ready. Allowing SVSM to operate in multiple threads may help mitigate the impact of these factors and improve migration speed.

The maximum allowed downtimes required for the successful migration of confidential machines show a higher relative increase compared to the migration speed slowdown. We expect this to be due to the fact that slower migration speed also

means slower convergence, while the migration time limit was the same in both cases.

An example: If a confidential machine reduces its dirty pages by 5.000 each iteration but each iteration takes 7.4 times longer than on a non-confidential machine, the confidential machine may miss the migration deadline (ten minutes) as it completes far fewer iterations. The non-confidential machine, with much faster iterations, can perform more rounds and finish migrating on time. The number of iterations corresponds to the number of dirty-page synchronizations performed, since dirty-page synchronization occurs at the very beginning and after each iteration. The average number of iterations is summarized in Table 6.3.

Apart from showing the similarity between the two variables, the Figure 6.1 provides insight into how close the measured minimal allowed downtime is to the real minimal allowed downtime. The longer the total migration time, while still successfully completing before the deadline, the closer we are approaching the real shortest allowed downtime. For instance, in the non-confidential case with a 50 MB/s writing speed, the machine migrated in under 1 minute, whereas the total migration time limit was set to 10 minutes. This suggests that decreasing the step size may yield a slightly better solution. In contrast, for writing speeds of 400 MB/s and 500 MB/s in the confidential case, the migration took over 7 minutes on average, indicating that the reported value is closer to the actual minimal allowed migration time.

The total time itself is not an interesting result of our experiments, as it is influenced by the allowed downtime. By increasing the maximum downtime, we can achieve shorter total migration times.

6.4 Summary

This chapter addresses research question R3, highlighting notable performance differences between confidential and non-confidential machines. Confidential machines exhibited substantially slower **RAM** migration speeds, with times 7.4 and 7.2 times slower compared to their non-confidential counterparts. Similarly, the required downtime for reliable migration was significantly longer, ranging from 9 to 15 times longer in the first, more controlled environment and 27 to 57 times longer in a real-world scenario.

# conn	NC Down (ms)	C Down (ms)	NC speed (pps)		C speed (pps)		NC total pages	C total pages
			Median	Mean \pm SD	Median	Mean \pm SD		
10	300	8,000	33,510	33,777 \pm 526	4,408	4,393 \pm 21	208,412	1,257,606
50	300	8,000	33,240	33,158 \pm 157	4,415	4,393 \pm 34	216,661	1,102,900
300	300	16,000	33,080	33,178 \pm 246	4,568	4,431 \pm 265	214,048	1,292,323
400	300	13,000	33,100	33,278 \pm 358	4,649	4,431 \pm 334	218,684	1,005,258
600	300	16,000	32,000	32,996 \pm 26	4,799	5,039 \pm 336	214,646	1,022,921
2000	300	17,000	33,160	33,180 \pm 206	5,053	5,039 \pm 25	212,439	1,021,352
5000	300	17,000	33,070	33,160 \pm 225	4,577	4,469 \pm 278	211,798	1,153,992
average	—	—	33,100	33,249 \pm 169	4,638	4,431 \pm 301	—	—

Table 6.2: Migration performance with different numbers of connections. The maximum allowed downtimes for non-confidential (NC) and confidential (C) machines are specified in milliseconds (ms).

Load (MB/s)	Non-confidential	Confidential
10	162	41
15	15	37
20	764	159
30	1040	40
50	51	97
75	214	37
100	86	123
200	244	19
300	201	23
400	149	49
500	29	24
median	162	40

Table 6.3: The number of dirty page synchronizations performed. This corresponds to the number of iterations performed in [QEMU](#). The symbol * indicates outliers.

7. Conclusion

In this thesis, we addressed the question of what is required for live migration to be implemented for [AMD SEV-SNP](#). We presented a design and implementation advancement for live migration of [confidential virtual machine](#) based on [AMD SEV-SNP](#) and the [Coconut-SVSM](#). Following feedback on our initial draft from the community, we based our solution on the existing [QEMU/KVM](#) migration framework and modified it to support live migration of [confidential virtual machine](#).

We examined the architecture of [Coconut-SVSM](#) and integrated a migration handler that cooperates with [QEMU](#) to securely transfer guest memory state. The design ensures that confidential memory pages are encrypted and integrity-protected before leaving the trusted domain, while non-confidential state continues to be migrated using existing [QEMU](#) mechanisms. A bitmap-based tracking mechanism for validated pages was implemented, and an experimental approach was taken to enable dirty-page tracking for guest memory backed by *guest_memfd*, despite the lack of official support.

The security analysis highlighted potential attack vectors, particularly the reliance on hypervisor-provided dirty-page tracking, and we proposed mitigation strategies based on cumulative hashing of migrated pages. Finally, we outlined the need for a two-phase checkpointing mechanism to guarantee that no guest [vCPUs](#) execute after migration is finalized, thereby preventing data consistency attacks.

Our work demonstrated that live migration of confidential virtual machines is feasible without introducing new migration commands. At the same time, we identified several challenges that remain open, ranging from interaction between [SVSM](#) and hypervisor to limitations of current [KVM](#) interfaces for confidential computing.

Finally, we evaluated our single [vCPU](#) solution using two benchmarks and estimated the [RAM](#) migration to be 7.4 times slower than using the fine-tuned migration of non-confidential machines. As [RAM](#) typically represents the largest resource, we expect this to closely approximate the slowdown of live machine migration if our solution were to be extended.

Overall, this work contributes a design, implementation strategy, and security analysis that can serve as a foundation for future development of live migration support in [Coconut-SVSM](#) and similar confidential computing platforms.

8. Future Work

While the design and prototype presented here provide a solid foundation, a number of important directions remain for future work:

- **Extended attestation mechanisms.** Our design establishes a secure channel between the source and destination [SVSM](#) instances, but it does not address the continuity of the migration chain. A critical aspect that requires further research is the ability for the guest owner to verify that the machine is running on a genuine confidential platform even after migration.
- **Robust notification mechanism.** Currently, the migration handler in [SVSM](#) is launched on an extra [vCPU](#) that busy-waits for migration to start. Future work should explore interrupt-based or event-driven mechanisms that allow efficient and scalable notification without relying on additional static [vCPUs](#).
- **Support for post-copy migration.** Our design implements pre-copy (push-based) migration with a final stop-and-copy phase. Post-copy migration, where execution resumes at the destination before all pages are transferred, could significantly reduce downtime.
- **Swapped-out and shared pages.** Our work does not address migration of swapped-out guest pages or pages marked as shared between [VMs](#). Extending migration to handle these cases securely is necessary for production use.
- **Extensible per-page metadata.** We implemented a bitmap to track validated pages. A more general tracking structure supporting compression, multiple attributes (e.g., access rights, shared/private status), and runtime reallocation would improve scalability for large [VMs](#).
- **Integration with [SVSM](#) user space.** Since [SVSM](#) user space is not yet implemented, our prototype added all functionality to the [SVSM](#) kernel. A natural next step is to move services such as the migration handler into user space, reducing trusted computing base size and improving maintainability.
- **Cross-architecture support.** Although this work targeted [AMD SEV-SNP](#), future extensions could adapt the design to [Intel TDX](#) or other confidential computing technologies, improving portability of migration mechanisms.

Addressing these challenges would bring migration of confidential VMs closer to practical deployment, enabling cloud providers and enterprises to leverage confidentiality without sacrificing flexibility or availability.

Bibliography

- [1] Dominic P. Mulligan, Gustavo Petri, Nick Spinale, Gareth Stockwell, and Hugo J. M. Vincent. Confidential Computing - a brave new world. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED), Washington, DC, USA, September 20-21, 2021*, pages 132–138. IEEE, 2021.
- [2] Confidential Computing Consortium et al. A Technical Analysis of Confidential Computing, 2022.
- [3] Xun Yi, Russell Paulet, and Elisa Bertino. *Homomorphic Encryption*, pages 27–46. Springer International Publishing, Cham, 2014.
- [4] Chuan Zhao, Shengnan Zhao, Minghao Zhao, Zhenxiang Chen, Chong-Zhi Gao, Hongwei Li, and Yu-an Tan. Secure Multi-party Computation: Theory, practice and applications. *Inf. Sci.*, 476:357–372, 2019.
- [5] International Organization for Standardization. Information technology — TPM Library — Part 1: Architecture, 2015.
- [6] Mengyuan Li, Yuheng Yang, Guoxing Chen, Mengjia Yan, and Yinqian Zhang. SoK: Understanding Design Choices and Pitfalls of Trusted Execution Environments. In Jianying Zhou, Tony Q. S. Quek, Debin Gao, and Alvaro A. Cárdenas, editors, *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2024, Singapore, July 1-5, 2024*. ACM, 2024.
- [7] Mingliang Pei, Hannes Tschofenig, Dave Thaler, and David Wheeler. Trusted Execution Environment Provisioning (TEEP) Architecture. *RFC*, 9397:1–31, 2023.
- [8] Henk Birkholz, Dave Thaler, Michael Richardson, Ned Smith, and Wei Pan. Remote ATtestation procedureS (RATS) Architecture. *RFC*, 9334:1–46, 2023.
- [9] Jana Eisoldt, Anna Galanou, Andrey Ruzhanskiy, Nils Küchenmeister, Yewgenij Baburkin, Tianxiang Dai, Ivan Gudymenko, Stefan Köpsell, and Rüdiger Kapitza. SoK: A cloudy view on trust relationships of CVMs - How Confidential Virtual Machines are falling short in Public Cloud. *CoRR*, abs/2503.08256, 2025.
- [10] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual

- Machines. In Amin Vahdat and David Wetherall, editors, *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings*. USENIX, 2005.
- [11] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. Post-copy live migration of virtual machines. *ACM SIGOPS Oper. Syst. Rev.*, 43(3):14–26, 2009.
- [12] William Voorsluys, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation. In Martin Gilje Jaatun, Gansen Zhao, and Chunming Rong, editors, *Cloud Computing, First International Conference, CloudCom 2009, Beijing, China, December 1-4, 2009. Proceedings*, volume 5931 of *Lecture Notes in Computer Science*, pages 254–265. Springer, 2009.
- [13] Haikun Liu, Cheng-Zhong Xu, Hai Jin, Jiayu Gong, and Xiaofei Liao. Performance and energy modeling for live migration of virtual machines. In Arthur B. Maccabe and Douglas Thain, editors, *Proceedings of the 20th ACM International Symposium on High Performance Distributed Computing, HPDC 2011, San Jose, CA, USA, June 8-11, 2011*, pages 171–182. ACM, 2011.
- [14] Changyeon Jo, Youngsu Cho, and Bernhard Egger. A machine learning approach to live migration modeling. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*, pages 351–364. ACM, 2017.
- [15] Fritz Alder, Arseny Kurnikov, Andrew Paverd, and N. Asokan. Migrating SGX Enclaves with Persistent State. *CoRR*, abs/1803.11021, 2018.
- [16] David Kaplan, Jeremy Powell, and Tom Wolle. SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. <https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf>, Oct 2021. Accessed: 2025-06-20.
- [17] David Kaplan. Protecting VM Register State with SEV-ES. <https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/Protecting-VM-Register-State-with-SEV-ES.pdf>, Feb 2017. Accessed: 2025-06-20.
- [18] Advanced Micro Devices, Inc. SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. <https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthe>

- [ning-vm-isolation-with-integrity-protection-and-more.pdf](#), Jan 2020. Accessed: 2025-06-20.
- [19] Hans Niklas Jacob, Christian Werling, Robert Buhren, and Jean-Pierre Seifert. *faultTPM: Exposing AMD fTPMs' Deepest Secrets*. In *8th IEEE European Symposium on Security and Privacy, EuroS&P 2023, Delft, Netherlands, July 3-7, 2023*, pages 1128–1142. IEEE, 2023.
- [20] Advanced Micro Devices, Inc. AMD64 Architecture Programmer's Manual Volume 2. https://docs.amd.com/v/u/en-US/24593_3.43, Mar 2024. Accessed: 2025-06-20.
- [21] Advanced Micro Devices, Inc. SEV-ES Guest-Hypervisor Communication Block Standardization. <https://docs.amd.com/v/u/en-US/56421>, Jan 2025. Accessed: 2025-06-20.
- [22] David Kaplan. Hardware VM Isolation in the Cloud. *Commun. ACM*, 67(1):54–59, 2024.
- [23] Advanced Micro Devices, Inc. Secure VM Service Module for SEV-SNP Guests. https://docs.amd.com/api/khub/documents/A15Q~fS1~kWKkJ3zC0vk_g/content, Jul 2023. Accessed: 2025-07-01.
- [24] Advanced Micro Devices, Inc. SEV Secure Nested Paging Firmware ABI Specification. <https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56860.pdf>, May 2025. Accessed: 2025-06-20.
- [25] COCONUT-SVSM Project Governance. <https://github.com/coconut-svsm/governance/tree/main>. Accessed: 2025-06-30.
- [26] Microsoft Corporation. *igvm_defs* - crates.io: Rust Package Registry. https://crates.io/crates/igvm_defs, 2025. Accessed: 2025-06-20.
- [27] Peter Xu. KVM Dirty Ring Interface. https://static.sched.com/hosted_files/kvmforum2020/97/kvm_dirty_ring_peter.pdf. Accessed: 2025-08-15.
- [28] Masanori Misono, Dimitrios Stavrakakis, Nuno Santos, and Pramod Bhatotia. Confidential VMs Explained: An Empirical Analysis of AMD SEV-SNP and Intel TDX. *Proc. ACM Meas. Anal. Comput. Syst.*, 8(3):36:1–36:42, 2024.
- [29] Jinyu Gu, Zhichao Hua, Yubin Xia, Haibo Chen, Binyu Zang, Haibing Guan, and Jinming Li. Secure Live Migration of SGX Enclaves on Untrusted Cloud.

In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*, pages 225–236. IEEE Computer Society, 2017.

- [30] Intel Corporation. Trust Domain Extension (TDX) Migration TD Design Guide. <https://cdrdv2.intel.com/v1/dl/getContent/733580>, Dec 2023. Accessed: 2025-06-20.
- [31] Intel Corporation. rust-migtd. <https://github.com/intel/MigTD>. Accessed: 2025-06-30.
- [32] Paolo Bonzini. The Confidential Computing Story part i: Rivers, dams and kernel development. https://gitlab.com/qemu-project/kvm-forum/-/raw/main/_attachments/2024/kvmforum24-tdx_GhPzxFo.pdf, 2024. Accessed: 2025-06-20.
- [33] The QEMU Project Developers. AMD Secure Encrypted Virtualization (SEV) - QEMU documentation. <https://www.qemu.org/docs/master/system/i386/amd-memory-encryption.html>. Accessed: 2025-06-20.
- [34] Dov Murik. [RFC PATCH v2 00/12] Confidential guest-assisted live migration. <https://lore.kernel.org/qemu-devel/20210823141636.65975-1-dvmurik@linux.ibm.com/>, Aug 2021. Accessed: 2025-06-20.
- [35] Tobin Feldman-Fitzthum. [RFC PATCH 0/9] Firmware Support for Fast Live Migration for AMD SEV. <https://edk2.groups.io/g/devel/message/79517>, Aug 2021. Accessed: 2025-06-20.
- [36] Paolo Bonzini. KVM: guest_memfd() and per-page attributes. <https://lwn.net/Articles/950433/>. Accessed: 2025-06-20.
- [37] Chenhui Ji, Dingji Li, Zeyu Mi, and Yubin Xia. PrometheusMigrate: Efficient Live Migration of Confidential Virtual Machine with Software Abstraction. In *2024 IEEE International Conference on Joint Cloud Computing (JCC)*, pages 1–8, 2024.
- [38] Alhish Kalra. Live migration of confidential guests. https://lpc.events/event/11/contributions/958/attachments/769/1448/Live%20migration%20of%20confidential%20guests_LPC2021.pdf, Sep 2021. Accessed: 2025-06-20.
- [39] Pankaj Gupta, Tom Lendacky. SEV-SNP Live Migration and VMM/KVM API Implications. <https://lpc.events/event/17/contributions/1532/attachments/1369/2974/06%20LPC-SNP-Live-Migration.pdf>, Nov 2023. Accessed: 2025-06-20.

- [40] Pankaj Gupta. SNP Live Migration with guest-memfd & mirror VM. https://gitlab.com/qemu-project/kvm-forum/-/raw/main/_attachments/2024/SNP_Live_Migration_KVM_forum_2024_svDwxa3.pdf, Sep 2024. Accessed: 2025-06-20.
- [41] Ackerley Tng. [RFC PATCH 00/11] New KVM ioctl to link a gmem inode to a new gmem file. <https://lore.kernel.org/lkml/cover.1691446946.git.t.ackerleytng@google.com/>, Aug 2023. Accessed: 2025-06-20.
- [42] stress-ng (stress next generation). <https://github.com/ColinIanKing/stress-ng/>. Accessed: 2025-09-05.
- [43] wrk - a http benchmarking tool. <https://github.com/wg/wrk>. Accessed: 2025-06-20.
- [44] Ziqiao Zhou, Anjali, Weiteng Chen, Sishuai Gong, Chris Hawblitzel, and Weidong Cui. VeriSMo: A Verified Security Module for Confidential VMs. In Ada Gavrilovska and Douglas B. Terry, editors, *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 599–614. USENIX Association, 2024.
- [45] COCONUT Secure VM Service Module. <https://github.com/coconut-svsm/svsm>. Accessed: 2025-08-15.
- [46] Microsoft Corporation. VeriSMo: A formally verified security module for AMD confidential VMs. <https://github.com/microsoft/verismo>. Accessed: 2025-08-15.
- [47] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.*, 7(OOPSLA1):286–315, 2023.
- [48] Joerg Roedel. SUSE open-sources Secure VM Service Module for Confidential Computing. <https://www.suse.com/c/suse-open-sources-secure-vm-service-module-for-confidential-computing/>. Accessed: 2025-08-15.
- [49] Advanced Micro Devices, Inc. Linux SVSM (Secure VM Service Module). <https://github.com/AMDESE/linux-svsm/tree/main>. Accessed: 2025-06-30.
- [50] Carlos Bilbao. The Linux SVSM project. <https://lwn.net/Articles/921266/>. Accessed: 2025-06-30.

- [51] jshelby. COCONUT-SVSM Joins the Confidential Computing Consortium: Enhancing Security for Sensitive Workloads. <https://confidentialcomputing.io/2024/07/02/coconut-svsm-joins-theconfidential-computing-consortiumenhancing-security-for-sensitiveworkloads/>. Accessed: 2025-06-30.
- [52] COCONUT-SVSM Development Plan. <https://coconut-svsm.github.io/svsm/developer/DEVELOPMENT-PLAN/>. Accessed: 2025-06-30.
- [53] coconut-svsm/linux fork from torvalds/linux. <https://github.com/coconut-svsm/linux/tree/svsm>. Accessed: 2025-08-15.
- [54] Stefan Hajnoczi. Merge tag 'for-upstream' of <https://gitlab.com/bonzini/qemu> into staging. <https://gitlab.com/qemu-project/qemu/-/commit/b92b39af4219df4250f121f64d215506909c7404>. Accessed: 2025-08-15.
- [55] Michael Larabel. QEMU 9.1 Released With AMD SEV-SNP Support & Intel IAA Acceleration During VM Migrations. <https://www.phoronix.com/news/QEMU-9.1-Released>. Accessed: 2025-08-15.
- [56] coconut-svsm/qemu forked from qemu/qemu. <https://github.com/coconut-svsm/qemu>. Accessed: 2025-08-19.
- [57] Michael Larabel. Linux 6.14 Looks To Support AMD's Zen 5 RMPREAD Instruction & Segmented RMP Mode. <https://www.phoronix.com/news/Linux-6.14-AMD-RMPREAD>, 2024. Accessed: 2025-06-20.
- [58] Joerg Roedel. SVSM Development Call July 2nd, 2025. <https://lore.kernel.org/coconut-svsm/8f73b209-f521-4770-b940-7f6dff3accd5@amd.com/>. Accessed: 2025-08-17.
- [59] Jonathan Corbet. Guest-first memory for KVM. <https://lwn.net/Articles/949277/>. Accessed: 2025-08-17.
- [60] Sean Christopherson. [RFC] KVM: mm: fd-based approach for supporting KVM guest private memory. <https://lore.kernel.org/all/20210824005248.200037-1-seanjc@google.com/>. Accessed: 2025-08-17.
- [61] Chao Peng. [PATCH v10 0/9] KVM: mm: fd-based approach for supporting KVM. <https://lore.kernel.org/all/20221202061347.1070246-1-chao.p.peng@linux.intel.com/>. Accessed: 2025-08-17.

TRITA-EECS-EX-2026:4
Stockholm, Sweden 2026

www.kth.se