SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

CloakVM: A Runtime System for Confidential Serverless Functions

Michael Hackl

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

CloakVM: A Runtime System for Confidential Serverless Functions

CloakVM: Ein Laufzeitsystem für vertrauliche serverlose Funktionen

Author:	Michael Hackl
Examiner:	Prof. Pramod Bhatotia
Supervisors:	Patrick Sabanic
	Dr. Dimitrios Stavrakakis
Submission Date:	April 15, 2025

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, April 15, 2025 Michael Hackl

Abstract

Serverless computing offers a scalable model for deploying applications without requiring developers to manage infrastructure. However, traditional serverless platforms fall short when handling confidential workloads since the cloud provider controls the systems. Confidential computing with trusted execution environments (TEEs) mitigates this risk by isolating workloads from a potentially malicious host operating system or hypervisor. Nevertheless, straightforward integrations of TEEs into serverless platforms present trade-offs: using a separate TEE per function invocation creates significant startup overhead while running multiple functions in a single TEE compromises isolation.

This thesis presents *CloakVM*, a runtime system for confidential serverless functions that combines security and performance. CloakVM leverages AMD SEV-SNP to run a confidential virtual machine containing a small, trusted Monitor that isolates serverless functions and delegates non-sensitive operations to an untrusted guest operating system. Functions use a Zygote mechanism to reuse loaded execution environments across invocations and pass data efficiently through shared memory channels. To ensure confidentiality and integrity, CloakVM uses hardware-enforced isolation and remote attestation, which is optimized via incremental measurements and local verification to reduce performance overhead.

Our evaluation shows that CloakVM achieves significantly lower startup latency and communication overhead than naive approaches, making it well-suited for sensitive serverless workloads.

Contents

1 Introduction

Serverless computing is a cloud service that allows developers to focus on building applications using small functions while the cloud provider manages the infrastructure. However, such serverless computing platforms are unsuitable for security-sensitive workloads because the execution environment is fully controlled by the cloud provider.

While cloud platforms support encryption at rest and encryption in transit for serverless functions, these mechanisms do not protect data during execution. Once decrypted in memory for processing, sensitive data becomes exposed to the host operating system, hypervisor, or any compromised infrastructure components. This creates a trust issue: users must rely on the provider not to inspect or tamper with their code and data. This concern is especially relevant for use cases where strong confidentiality and integrity guarantees are required by law (e. g., protection of personal data or health information) or critical for the business (e. g., intellectual property, proprietary algorithms, or AI models).

Confidential computing addresses this concern by offering hardware-enforced trusted execution environments (TEEs) that ensure sensitive workloads remain confidential by isolating and protecting user workloads from the underlying infrastructure. They ensure that even the hypervisor and host OS cannot access the memory or CPU state of protected workloads. To prove that they are authentic and uncompromised, TEEs support a mechanism called remote attestation. This process allows users to cryptographically verify that their code is running inside an authentic TEE on trusted hardware before releasing secrets or sensitive data.

Integrating TEEs into serverless computing introduces several challenges. Serverless functions are typically short-lived: an analysis of the Azure Functions production workload found that more than half of all functions execute for less than 1 second on average [37], making them particularly sensitive to startup overhead. Naively launching a new TEE for each invocation introduces various overheads, such as significant startup and attestation latency, while reusing a single TEE across invocations compromises isolation and complicates attestation.

In this thesis, we present *CloakVM*, a runtime system for efficient and secure execution of confidential serverless functions. Built on AMD SEV-SNP, CloakVM runs multiple functions within a single TEE while preserving isolation using hardware privilege levels and address space separation. It has a small trusted computing base consisting of only a lightweight trusted Monitor. To reduce startup latency, CloakVM reuses pre-initialized execution environments and speeds up attestation by decoupling environment verification from function verification. It further supports efficient, confidential function chaining through direct memory sharing, eliminating the need for costly re-encryption or re-attestation across function boundaries.

The remainder of this thesis is organized as follows:

- Chapter 2: Background
  Introduces serverless computing and confidential computing technologies, with a focus on AMD SEV-SNP.
- Chapter 3: Overview
  Outlines the motivation and goals of CloakVM, along with its system architecture and workflow.
- Chapter 4: Design
  Provides a discussion of the security properties and performance optimizations that CloakVM provides.
- Chapter 5: Implementation
  Describes the implementation of CloakVM's components and their interactions in detail.
- Chapter 6: Evaluation
  Evaluates the performance of CloakVM through microbenchmarks and application-level benchmarks.
- Chapter 7: Related Work
  Presents existing approaches related to serverless confidential computing.
- Chapter 8: Conclusion and Future Work
  Summarizes this thesis and concludes with an outlook on how the system could be extended and improved.

2 Background

2.1 Serverless Computing

Cloud computing enables users to provision and manage computing resources over the internet, offering dynamic scaling and flexible pricing models. With cloud providers, such as Amazon Web Services (AWS) [11], Microsoft Azure [34], and Google Cloud Platform (GCP) [21], responsible for the physical and virtual infrastructure, the user does not need to invest in and maintain local infrastructure, allowing them to focus on the development of applications.

One popular cloud service is "serverless functions," also referred to as Function as a Service (FaaS) [13, 33, 19]. Serverless computing allows developers to run code without being concerned about individual servers or their setup, as the cloud provider automatically provisions them based on demand and configures the OS and runtime environment for the code. Users are only billed for the actual resources consumed. Thanks to their scalability and pay-per-use pricing, serverless functions are well-suited for workloads with unpredictable and intermittent invocations.

After a *function provider* (e. g., developer) uploads his code to the FaaS provider, it can be invoked by *function callers* (e. g., an application client) through triggers, e. g., an HTTP request from a frontend app or a timer. If a function was recently invoked and remains loaded in memory, it can start quickly — this is called a warm start. In contrast, if the code must be fetched from storage, it is called a cold start.

To build more complex workflows, multiple serverless functions can be orchestrated and linked together in a process called function chaining, where the output of one function becomes the input of the next. This allows developers to design applications that involve multiple steps, conditional branching, and parallel execution.

In private clouds, where the infrastructure is dedicated to a single organization, serverless functions are often isolated using container runtimes like Docker with runC. This approach provides a lightweight process and namespace isolation, which, while it can be sufficient for trusted workloads within an organization, lacks strong security boundaries against sophisticated attacks. That is why major public cloud providers use stronger isolation mechanisms, such as AWS's micro virtual machines (Firecracker) [10] and Google's user space kernel that restricts system calls (gVisor) [15].

2.2 Confidential Computing

We provide an overview of confidential computing and trusted execution environments (TEEs) before explaining the specific implementation that we use to implement CloakVM, AMD Secure Encrypted Virtualization with Secure Nested Paging (SEV-SNP), in detail.

2.2.1 Overview

Confidential computing addresses scenarios where the workload and the infrastructure are managed by different parties, ensuring data privacy and security. While traditional virtualization allows cloud providers to protect their servers from potentially malicious users, confidential computing extends this protection in the other direction to protect users from potentially malicious cloud providers.

These protections are beneficial even in benign cases, as vulnerabilities can be exploited if a system becomes compromised. A system's trusted computing base (TCB), i. e., the hardware, firmware, and software components critical to its security, should be minimized to reduce its attack surface. Confidential computing achieves this by removing the host system and its hypervisor from the TCB. A smaller TCB also allows for easier verification and better maintainability through reduced complexity.

Confidential computing is implemented in the CPU hardware through TEEs. TEEs provide confidentiality, i. e., preventing unauthorized read accesses to code and data, and integrity, i. e., preventing unauthorized modifications to code and data. They also provide cryptographic proof that a specific workload is executing inside an authentic TEE through a process called attestation.

There are two categories of TEEs: process-based TEEs, such as Intel Software Guard Extensions (SGX) [24], which protect selected code segments with enclaves, and VM-based TEEs, such as AMD SEV-SNP [2] and Intel Trust Domain Extensions (TDX) [25], which protect entire virtual machines (VMs), which are then called confidential virtual machines (CVMs).

Although TEEs mitigate many software-based attack vectors and some hardware-based attacks (e. g., cold-boot attacks), current technologies do not provide full protection against all physical or side-channel attacks. Also, while the cloud provider does not need to be trusted, trust is still required in the CPU manufacturer, as it controls the implementation and security of the TEE.

Confidential computing is of interest to anyone requiring privacy or protection of secrets, e. g., the healthcare and finance industries, among others.

2.2.2 AMD SEV-SNP

AMD SEV-SNP [3, 27, 26] is available on 3rd generation and later AMD EPYC processors [12, 32, 20]. In addition to VM isolation, it provides two additional isolation mechanisms.

Hypervisor-Guest Isolation

AMD SEV-SNP implements a TEE by encrypting the memory used by CVMs. This memory is encrypted with different keys for different CVMs and is only decrypted on-the-fly within the CPU. The AMD Secure Processor (AMD-SP), a coprocessor dedicated to security tasks, generates and stores these encryption keys — ensuring that they are inaccessible to software, including the hypervisor.

When a CVM's virtual central processing unit (vCPU) is paused, its register state is automatically stored in an encrypted Virtual Machine Save Area (VMSA), protecting it from inspection or modification by the hypervisor.

In addition to encryption, AMD SEV-SNP ensures that memory content is protected from unauthorized modifications. The Reverse Map Table (RMP) enforces this integrity protection by tracking memory ownership (either the hypervisor or a specific guest) and mappings to prevent the hypervisor from overwriting memory that is owned by the guest or changing guest page table entries (guest physical address (GPA) to system physical address (SPA) mappings) without informing the guest. The hypervisor can modify the RMP only through the special `RMPUPDATE` instruction, and the guest must validate each change using the special `PVALIDATE` instruction before it can use the corresponding memory location. After each address translation,[1] the CPU checks

[1]For read accesses from the hypervisor, this check is skipped because the memory encryption already ensures confidentiality.

the owner and ensures that the page table and the RMP mappings agree, granting or denying memory access based on the result.

To start a CVM, the initial state of code, data, and CPU configuration is provided to the AMD-SP, which cryptographically measures these (i. e., computes a cryptographic hash) and encrypts them for the CVM. The AMD-SP is also responsible for generating a cryptographically signed attestation report including these measurements, allowing users to verify the integrity of the system's hardware and software.

For communication with non-encrypted components, guests can specify whether certain memory pages are encrypted or unencrypted by setting a specific bit (called C-bit or enCrypted bit) in the page table entries. One such unencrypted page is the Guest-Hypervisor Communication Block (GHCB) [8], which is used to exchange information (e. g., hypercall arguments and responses) with the hypervisor.

Intra-Guest Isolation

In addition to isolation between the host and the guest system, AMD SEV-SNP also provides a mechanism for isolation within a CVM by introducing Virtual Machine Privilege Levels (VMPLs). There are four VMPLs, with level 0 being the most privileged and level 3 the least privileged. All VMPLs share the same guest physical address space, but each VMPL maintains its own VMSA that holds the vCPU state for that particular privilege level. A vCPU running at a specific VMPL is subject to the memory access rights (read, write, supervisor-mode execute, and user-mode execute) associated with this level, which are stored in the RMP and can be changed using the special `RMPADJUST` instruction from a higher-privileged VMPL. When these rights are more restrictive than page-table permissions, they take precedence.

VMPLs can be leveraged to implement Secure VM Service Modules (SVSMs): software in VMPL0 that provides secure services, such as a virtual Trusted Platform Module (vTPM), to the guest. (The hypervisor-emulated vTPM cannot be used in the AMD SEV-SNP security model as the hypervisor is untrusted.)

3 Overview

3.1 Motivation and Goals

CloakVM aims to enable the secure execution of serverless functions that process sensitive data in untrusted cloud environments. A straightforward approach for confidential serverless functions is to execute the functions within TEEs. However, naive implementations suffer from drawbacks:

Function Execution Models

One TEE per function execution. This approach ensures strong isolation between functions but suffers from long startup delays (quantified in Subsection 6.2.1). Furthermore, chaining multiple functions is inefficient since the data must cross multiple encryption boundaries (quantified in Subsection 6.2.3).

A single TEE for multiple function executions. While this model reduces startup overhead, it sacrifices the strong isolation between functions. Additionally, attestation becomes less precise: Since the TEE persists across multiple function executions, a single attestation report only proves that the TEE was correctly initialized — not which specific code was running during a request.

TEE Implementations

CVMs. Running a CVM with the typical Linux OS results in a relatively large TCB and therefore also attestation time (quantified in Section 6.3).

Enclaves. Intel SGX enclaves offer a smaller TCB but do not allow shared unencrypted memory with the host, prohibiting efficient memory reuse and leading to increased communication overhead.

Overall, each combination of these execution models and TEE technologies is not well-suited for confidential serverless functions: either due to security or performance reasons.[1]

CloakVM addresses these shortcomings by providing:

- **Confidential and isolated function execution,** which protects sensitive data (Subsection 4.1.1).
- **A small TCB,** which lowers the attack surface (Subsection 4.1.2).
- **Fast function startup,** which improves responsiveness for user requests (Subsection 4.2.1).
- **Efficient function chaining,** which minimizes overhead and makes complex workflows more efficient (Subsection 4.2.2).

3.2 Structure and Workflow

To achieve its goals, CloakVM combines the strengths of the aforementioned options while addressing their shortcomings: CloakVM uses the "single TEE for multiple function executions" model but maintains isolation via traditional protection rings and AMD SEV-SNP VMPLs. It runs in a CVM but removes the OS from the TCB.

The *Monitor* is the core component of CloakVM and is responsible for the secure execution of serverless functions. It runs as a SVSM in a CVM at VMPL0. During initialization, the function provider verifies the Monitor. As part of this remote attestation process, the Monitor and the function provider establish a secure channel. Through this channel, the function provider configures the Monitor with a decryption key for the function inputs called *Function Key* and a *Policy* specifying for which functions the key is allowed to be used.

For general management tasks, such as running a serverless orchestration framework (e. g., Apache OpenWhisk [14] or Knative [38]), networking, and storage, the Monitor launches a conventional, untrusted *Guest OS* (e. g., Ubuntu) isolated in a lower-privilege VMPL.

[1]Without security, confidentiality cannot be guaranteed. Bad performance is problematic because many serverless functions have such short execution times that the overhead becomes noticeable and negatively impacts the user experience [37].

In CloakVM, a serverless function consists of a Zygote and the function code itself. A *Zygote* (named after the Android counterpart) is a template process that contains the function execution environment. To improve performance, instead of creating a new execution environment for every function invocation, we reuse already loaded and initialized templates. Zygotes can include runtimes (e. g., Python or Node.js) and function dependencies.

When the function code is combined with the specified Zygote, they form a *Trustlet*. Trustlets are processes that the Monitor executes — securely isolated from other Trustlets and the Guest OS. Trustlets can be chained together so that the output of one Trustlet is the input of another.

When a (encrypted) function request arrives, CloakVM first checks whether the necessary Zygote and function code are already loaded. If they are not, the Guest OS fetches the Zygote image and function code from a registry. The Monitor then loads each component, measures it, verifies it against the Policy, and initializes the Zygote. After that, the Monitor combines the Zygote with the function code to create a Trustlet (for every chain link when functions are chained). On subsequent requests for the same function, CloakVM can skip these steps by reusing the already initialized components. Finally, the Guest OS forwards the function input to the Monitor, which decrypts it and executes the function in an isolated environment.

If the function requires functionality that the Trustlet cannot provide (e. g., file I/O), the Trustlet sends a request to the Monitor, which forwards it to the Guest OS when appropriate.

Once the function completes, the Monitor encrypts the result and returns it, along with a Monitor-generated attestation report[2] that allows the function caller to verify the integrity and authenticity of the execution.

Figure 3.1 shows an overview of CloakVM's components, where each important trust boundary is labeled with the corresponding isolation mechanism. Subsection 4.1.1 describes these mechanisms in detail.

[2]This report is generated and signed by the Monitor, not by the AMD-SP. It is trusted because the Monitor itself was verified via AMD SEV-SNP attestation beforehand.

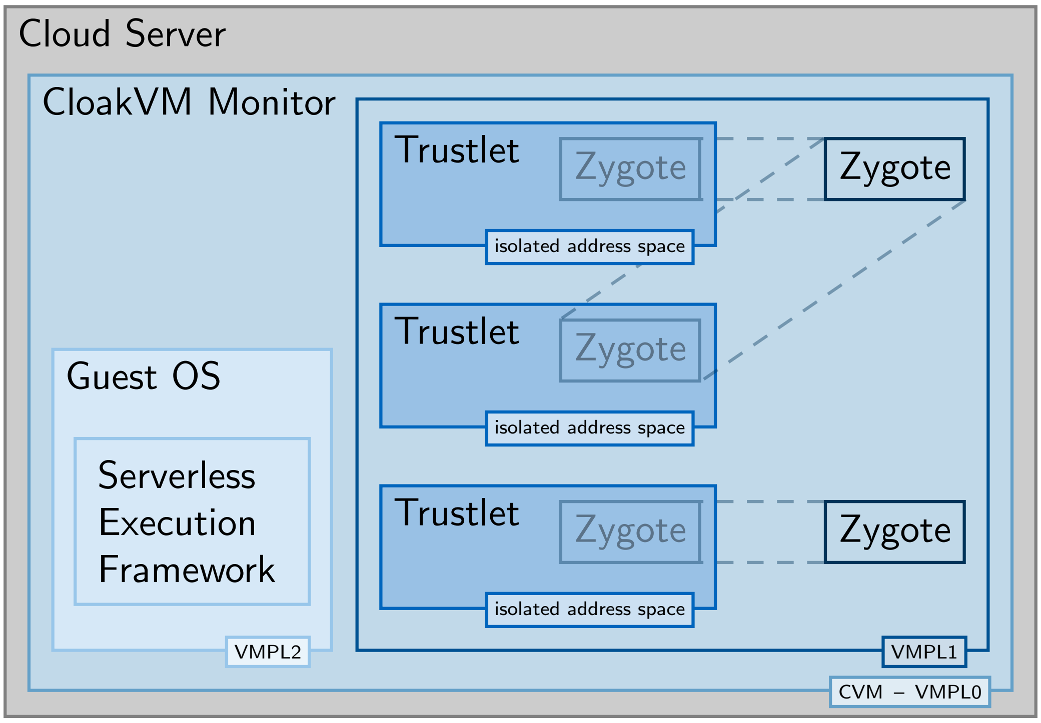

Figure 3.1: CloakVM architecture.

4 Design

In this chapter, we detail the design of CloakVM. We first discuss the security aspects of our system, followed by the performance optimizations implemented to address the challenges of confidential serverless functions.

4.1 Security

CloakVM provides security by ensuring confidentiality and integrity while minimizing the TCB.

4.1.1 Confidentiality and Integrity

Besides availability — which CVMs cannot guarantee[1] — confidentiality and integrity are two core pillars of information security that CloakVM addresses using remote attestation and multiple isolation mechanisms (see the labels in Figure 3.1).

Attestation. During initialization, the Monitor requests a signed attestation report of the CVM's memory, CPU state, and system information from the AMD-SP. By checking the attestation report's contents and verifying that its signature is anchored to the AMD root CA, this attestation report provides proof that the Monitor is authentic and executing within an authentic TEE.

When the Monitor is attested and verified, it can be trusted to perform secure measurements of other components such as Zygotes, Trustlets, and function inputs and outputs. These measurements are included in the function result as part of a Monitor-signed

[1]A malicious cloud provider or hypervisor can always shut down hardware resources. Potential mitigations include adopting a multi-cloud strategy or moving to a private (on-premises) cloud.

attestation report and are linked to the original AMD-SP attestation report.[2] In this way, CloakVM extends the chain of trust: AMD Root Key $\rightarrow$ *intermediate certificates from AMD*[3] *[1]* $\rightarrow$ attestation report by AMD-SP $\rightarrow$ Monitor $\rightarrow$ Zygote, Trustlet, function input, and function output.

When receiving the result, the function caller compares the measurements in the Monitor attestation report with the expected values to ensure that the correct function (i. e., the intended Zygote and function code) was executed with the correct input by CloakVM inside a secure CVM.

The Guest OS is not attested because it is considered untrusted.

Hypervisor and Host Isolation. AMD SEV-SNP protects the entire CVM (and thus CloakVM) from the hypervisor, host OS, and other VMs by enforcing hardware-based memory and CPU register encryption and integrity (see Subsection 2.2.2, *Hypervisor-Guest Isolation*).

Although all network traffic passes through the host, it remains confidential and tamper-proof due to encryption.

Trustlet Isolation. Since CloakVM uses the “single TEE for multiple function executions” model, we need to ensure isolation through another mechanism: Trustlets run in user-mode (in VMPL1[4]) and are isolated by their separate address spaces, which the Monitor manages via distinct page tables. This ensures that Trustlets cannot access memory belonging to other Trustlets, the Guest OS, or the Monitor.

CloakVM creates separate Trustlet instances for different function callers to protect against exploitation of the runtime environment through malicious input, which could allow an attacker to exfiltrate or manipulate information.

[2]This link is established through the Function Key: the function caller trusts this key from the function provider, who only transmits it to the Monitor after verifying the AMD-SP attestation report. This process is described in Subsection 4.2.1, *Attestation*.

[3]AMD Root Key $\rightarrow$ AMD SEV Key $\rightarrow$ Versioned Chip Endorsement Key (VCEK) $\rightarrow$ attestation report

[4]While VMPL separation from the Monitor is not strictly necessary for security, it provides an additional layer of defense in depth.

Guest OS Isolation. The Guest OS runs in VMPL2, which prevents it from accessing memory reserved for the Monitor and Trustlets running in higher-privileged VMPLs. From CloakVM's perspective, the Guest OS and the serverless orchestration framework belong to the same trust domain.

They cannot provide unauthorized serverless functions to the Monitor (e. g., to exfiltrate input data) because the Monitor measures and validates all functions. Also, the Guest OS can neither access input data (because it is encrypted and only the Monitor can decrypt it) nor access or manipulate output data (because it is encrypted and includes an signed attestation report).

4.1.2 Small Trusted Computing Base

A small TCB reduces the attack surface, simplifies verification, and improves maintainability.

We assume that the cloud provider and its hypervisor are untrusted, meaning an attacker could potentially control the entire host system except for the AMD System-on-Chip (SoC) hardware and the AMD-SP [3]. CloakVM's advantage over a typical Linux CVM is that it excludes the Guest OS from its TCB as well. Even if the Guest OS is compromised, CloakVM ensures that it cannot read or tamper with confidential data.

Only the Monitor and the selected Zygote are considered trusted. The Monitor is significantly smaller and less complex than a full operating system kernel. It delegates all non-security-sensitive tasks (e. g., networking, storage, function scheduling) to the untrusted Guest OS, while it only handles security-sensitive functions (e. g., key management, function attestation, and execution).

Furthermore, we increase the TCB's trustworthiness by using an attestation protocol that is formally verified to ensure secrecy, authenticity, and correctness [35], and by implementing the Monitor in Rust, a programming language that provides a strong type system and compile-time checks to reduce potential vulnerabilities.

4.2 Performance

CloakVM improves the efficiency of confidential serverless functions by addressing two performance bottlenecks: slow function startup and function chaining overhead.

4.2.1 Startup

Since serverless functions often have short execution times [37], the startup latency of the execution environment can represent a significant portion of the total runtime. CloakVM reduces this startup overhead by optimizing both environment initialization and attestation.

Initialization

CloakVM avoids the major startup overhead (see Subsection 6.2.1) of launching a new CVM for every function invocation by serving all function requests from a single CVM (running the Monitor). With this design, we not only avoid the CVM boot time, but we can also save other redundant computations and thereby reduce initialization time further:

With the Zygote mechanism, CloakVM loads the Zygote image, initializes it (library loading, runtime environment setup), and then reuses the resulting pre-initialized Zygote across multiple Trustlets. Similar to the Linux `fork()` system call, the Zygote is mapped read-only into the address space of each new Trustlet. To enable Trustlets to write to Zygote memory, the Monitor uses copy-on-write: when a Trustlet needs to alter a page, the Monitor creates a writable copy of the affected page specifically for this Trustlet. This strategy reduces both memory usage and startup delay for Trustlets when they can use a pre-initialized Zygote (while preserving their isolation). This improved resource utilization allows more Trustlets to be loaded concurrently within the same CloakVM instance.

Attestation

Attestation is essential for ensuring security, but it typically adds substantial latency (see Subsection 6.2.2). CloakVM mitigates this problem using two strategies: incremental attestation and local pre-attestation.

Incremental Attestation. Instead of re-measuring and re-attesting the entire execution environment for every function call, CloakVM adopts an incremental approach that reuses validated components across multiple function invocations.

The Monitor itself is attested at system startup. Zygote images and function code are measured the first time they are loaded. Their measurements are cached internally by the Monitor. On subsequent function executions, only the input and output need fresh measurement. All these measurements are combined to an attestation report that accompanies the function result. This report enables the function caller to confirm the integrity and origin of the result.

This strategy reduces latency, especially for frequently used Zygotes.

Local Attestation. CloakVM further improves performance by avoiding interactive remote attestation before each function execution. Instead, CloakVM attests functions locally using the Policy that the function provider has configured. To preserve security, the input to each function is encrypted using the Function Key that is provided to the Monitor only after the function provider has verified the Monitor. Because only a verified Monitor possesses the decryption key, and because the Monitor will only launch Trustlets approved by the Policy, this design ensures that confidential input data is never exposed to unauthorized or tampered code. As a result, function execution can begin immediately without requiring a remote attestation round trip.

Once execution completes, the Monitor uses the same key to append a signed attestation report to the output. This signature allows the function caller to verify that the attestation report — and therefore also the result — is untampered and originates from a function-provider-approved Monitor.

4.2.2 Function Chaining

Chaining functions across isolated TEEs incurs overhead due to re-encryption and re-attestation at every transition.

CloakVM avoids this overhead: because Trustlets execute within the same CVM and are isolated (only) by separate address spaces, the Monitor can map a shared memory region into the address spaces of two chained Trustlets. This way, the Monitor establishes a direct memory channel between these Trustlets, into which the first Trustlet writes its output and from which the subsequent Trustlet reads its input.

This design substantially reduces inter-TEE communication overhead, thereby making function chaining very efficient.

5 Implementation

This chapter explains how the security properties and performance optimizations presented in Chapter 4 are realized. We begin by describing the function creation and Monitor initialization, including remote attestation and Policy management, followed by a description of the function invocation and execution workflow. Figure 5.1 shows the typical workflow.

The current implementation focuses on the execution layer. Although integration with standard serverless orchestration frameworks is possible, we currently manage orchestration manually via a Python library running in the Guest OS.

5.1 System Setup

Before serverless functions can be invoked, three steps are required:

1. Packaging functions.
2. Launching the CloakVM Monitor.
3. Remote attestation and configuring the Policy.

5.1.1 Function Packaging

Zygote Images. A Zygote image contains the runtime environment. It consists of all resources necessary to execute a function, such as the language runtime (e. g., Python or Node.js), required system libraries, and dependencies. It serves as a template from which the actual Zygote will be created. Zygote images do not embed the function code in order to be reusable across multiple functions.

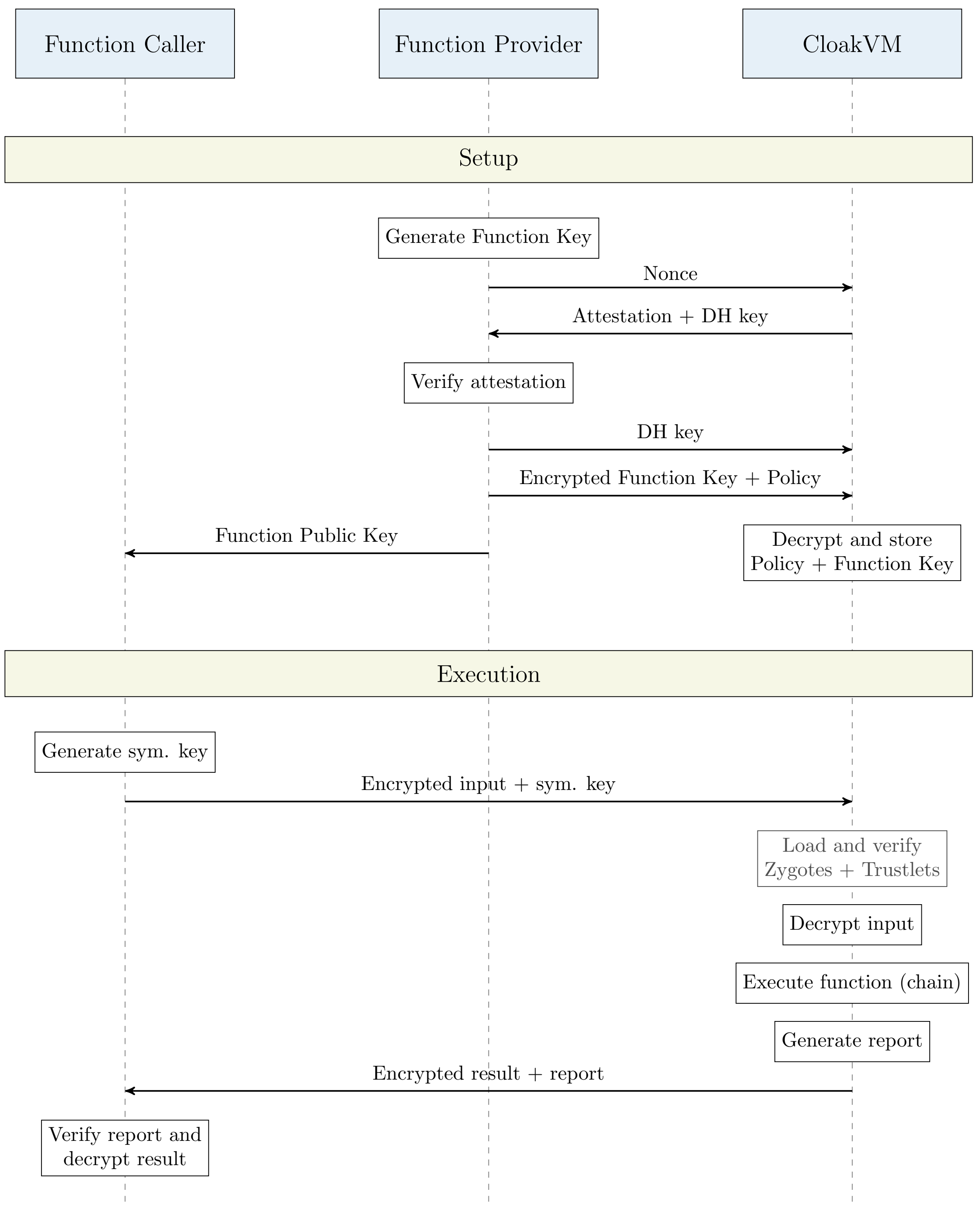

Figure 5.1: CloakVM workflow.

Function Code. Function authors only provide the application logic, whereas libraries and language runtimes are supplied by a potentially shared Zygote. Both components (Zygote images and function code) are stored in a registry from which the serverless orchestration framework can retrieve them on demand.

Gramine LibOS. Because the Guest OS is explicitly untrusted, we cannot rely on its system calls to provide functionality for the serverless functions. Instead, we embed a library operating system (LibOS) into the Zygote images: Gramine [22, 39, 40]. Gramine can execute unmodified Linux applications in isolated environments such as Intel SGX. It intercepts and handles system calls, allowing us to run normal user space applications without a regular OS [23].

The Gramine Platform Adaptation Layer (PAL) translates LibOS operations into system-specific operations and allows it to be ported to different backend environments. We implement a PAL that uses the Monitor to perform low-level operations like memory management.

Gramine Configuration. Each Zygote image includes a Gramine manifest that specifies how to load the function code, relevant environment variables, and file mounts.

While additional files can be fetched at runtime when requested by the function (see Subsection 5.2.3), having a local copy of essential libraries improves file access performance. For this reason, we compile an embedded filesystem directly into the Gramine binary. However, there is a trade-off: embedding more files increases the image size, which in turn slows down loading and attestation.

Example Zygote Image. For Python-based functions, the Zygote image bundles the Python interpreter, standard libraries, system libraries, and application-level dependencies such as NumPy. The image also contains a loader script that interfaces between the function code and the Monitor. This loader is responsible for loading the function code and input from the Monitor, calling the function, and returning the output to the Monitor.

5.1.2 Monitor Initialization

The Monitor is the core of CloakVM. It is implemented as a SVSM based on COCONUT-SVSM [16] running in a CVM. It operates at VMPL0 to protect it from lower-privileged components such as the Guest OS and Trustlets.

Monitor Responsibilities. The Monitor is responsible for securely booting, running, and isolating the Guest OS and functions. Its tasks are split between three main components:

- **Process Manager:** Interfaces with the Guest OS and is responsible for loading, initializing, and managing Zygotes and creating and managing the Trustlets based on them.
- **Process Runtime:** Provides the execution environment for Trustlets. It handles Monitor calls and their copy-on-write memory.
- **Attestation Service:** Measures Zygote images, function code, function inputs, and outputs, verifies Zygote images and function codes against their Policy and generates attestation reports.

CVM Boot Process. When the cloud platform boots the CVM, the AMD-SP measures the firmware, CVM memory, and configuration. These measurements are stored so they can later be used for the hardware-signed attestation report [7]. Once the CVM starts, the Monitor is the first software component, executing at VMPL0. Because memory validation (`PVALIDATE`) is slow, the Monitor pre-validates a set of memory pages at startup to reduce runtime overhead for subsequent allocations. After it completes its boot process, it allocates memory for the Guest OS with VMPL2 privileges and boots the Guest OS isolated in VMPL2. We use a special SVSM-compatible version of OVMF (Open Virtual Machine Firmware) and Ubuntu Linux [17] for this purpose.

Monitor–Guest OS Interface. The Monitor delegates all non-security-sensitive tasks (e. g., networking and serverless orchestration) to the untrusted Guest OS. It runs a serverless orchestration framework, which manages function triggers, downloads functions from the registry, and interacts with the Monitor to load Zygotes and Trustlets, create chains, execute them, and retrieve their results.

For communication with the Monitor, we extend the SVSM calling convention [6] with a new protocol for our Monitor calls (begin remote attestation, set up Function Keys and policies, create and delete Zygotes and Trustlets, set up channels, and invoke Trustlets).

1. The Guest OS places call parameters (e. g., a request to create a Zygote) into the CPU registers and the SVSM Calling Area (similar to the GHCB, but handled by the SVSM and encrypted for the hypervisor) [6].
2. The Guest OS instructs the hypervisor to switch the vCPU execution to VMPL0 (where the Monitor runs) by writing the `SNP Run VMPL` request to the GHCB and executing the hypercall with `VMGEXIT` [8].
3. The Monitor inspects the CPU registers (in the VMSA of VMPL2) and SVSM Calling Area and performs the requested action (e. g., copying a Zygote image from Guest OS memory).

We provide a Linux kernel module and user space library that wrap these interactions behind an API, simplifying the development of orchestration tools.

5.1.3 Remote Attestation and Policy Configuration

Finally, before any user workloads can run securely, the function provider must verify the Monitor and supply the Function Key and a Policy. By establishing trust once and permitting local enforcement of the Policy, the system achieves both strong security guarantees and efficient invocations.

Function Key. The function provider generates an asymmetric key pair known as the Function Key, which is used to encrypt function inputs and to sign attestation reports issued by the Monitor. The public part of this key is shared with all authorized function callers, enabling them to encrypt their input data. The private part remains confidential and is only transmitted to the Monitor after a successful attestation process.

The function provider must be trusted not to misuse the private key (e. g., to decrypt inputs that belong to a user). However, since the function provider controls the function code, this assumption is reasonable.

Policy. A Policy associates a Function Key with a function (or a sequence of functions, in the case of a chain) by specifying the approved Zygote image and function code measurement.

Secure Key and Policy Transfer. Before the function provider can share the private Function Key, it must first verify the authenticity and integrity of the Monitor. To start this process, the function provider begins a secure communication channel by generating a nonce (a random value) and sending it to the Monitor.[1]

In response, the Monitor returns the following:

- A Diffie-Hellman (DH) public key, which will be used to establish a shared secret for encrypted communication.
- A signed AMD SEV-SNP attestation report issued by the AMD-SP. This report includes the measurement of the CVM that the AMD-SP generated during CVM startup. It allows the function provider to verify the Monitor's integrity.
- Embedded within the attestation report: the nonce and a cryptographic hash of the DH public key. The nonce protects against replay attacks. The hash of the DH key prevents man-in-the-middle attacks by linking the key to the attestation report, thereby ensuring that it originates from the verified Monitor and has not been tampered with.

To obtain the attestation report, the Monitor uses a trusted communication channel provided by the AMD-SP. This channel is cryptographically protected against inspection or modification by the hypervisor. Specifically, the Monitor issues a `MSG_REPORT_REQ` via the `SNP_GUEST_REQUEST` hypercall [7]. This request includes user-specified data in the `REPORT_DATA` field (64 bytes), allowing the Monitor to embed the nonce and the DH key hash into the report. The attestation report is cryptographically signed by the AMD-SP, and its authenticity can be verified using AMD's certificate chain [1].

After verifying the attestation report and confirming the Monitor's identity, the function provider generates its own DH key pair and derives the shared secret using the Monitor's public key. It then responds with its own DH public key and — encrypted using the shared secret — the Function Key and Policy.

Using the function provider's public key, the Monitor derives the shared secret as well and decrypts the Function Key and Policy. Once the key and Policy are available, the Monitor is ready to accept confidential function inputs from function callers.

Because attestation and key provisioning occur per CloakVM instance, this process must be repeated whenever the cloud provider scales up the infrastructure. However, this process can be automated.

[1]All communication is relayed via the untrusted Guest OS.

5.2 Function Execution Workflow

Having established how CloakVM is initialized and how Zygote images and function code are prepared, we now describe the four major runtime steps:

1. Making a request.
2. Loading Zygotes and Trustlets.
3. Executing Trustlets.
4. Returning the result and its attestation report.

5.2.1 Invocation

First, the function caller obtains the public Function Key from the function provider. This key allows the caller to securely transmit data to the CloakVM Monitor.

Before calling the function, the caller generates a symmetric key. This symmetric key will be used by the Monitor to encrypt its response. The caller then encrypts both the function input data and this new symmetric key using the public Function Key.

Next, the caller sends the request (encrypted input data and symmetric key) to the serverless orchestration framework running on the untrusted Guest OS. The caller does not need to perform any verification: as described in Subsection 4.2.1 *Local Attestation*, if the Monitor has the private portion of the Function Key and can decrypt the input, it must have been verified by the function provider.

Upon receiving the request, the orchestration framework checks whether the necessary components (Zygotes and Trustlets) are already loaded in the Monitor. If one of them or both are not loaded, it downloads them from the registry and instructs the Monitor to load and verify them as described in the following subsection. If both components are already loaded, this step can be skipped, and the orchestration framework directly continues with Subsection 5.2.3, forwarding the encrypted request to the Monitor for execution.

5.2.2 Zygote and Trustlet Initialization

Zygote Image and Function Code Loading. When the Guest OS instructs the Monitor to prepare a new Zygote or a new Trustlet, the Monitor first copies the corresponding image or code from the Guest OS memory into Monitor-owned memory. Copying ensures that the untrusted Guest OS cannot modify the image once it is in the Monitor's control. The Monitor then measures the Zygote image or function code. This measurement is compared against the approved Zygote image–function pair(s) specified in the Policy of this function (or function chain). If the hash does not match, the loading process is aborted. This ensures that a compromised Guest OS or registry cannot provide unauthorized code.

If the measurement is valid, the Monitor caches it to avoid repeating the measurement for subsequent requests that reuse the same Zygote image or function code. Finally, the Monitor marks all pages for the Zygote image or function code as accessible only in VMPL1.

Zygote Initialization. Once the Zygote image has been validated and loaded, the Monitor initializes it by executing the startup code until it is ready to load a function. During this initialization phase, the runtime (e. g., Python or Node.js) is configured and brought to a ready state. This approach ensures that the time-consuming steps of starting up the runtime and loading system libraries are performed only once. After initialization is complete, the Zygote memory regions are marked read-only for the copy-on-write mechanism.

Trustlet Creation. To create a Trustlet, the Monitor creates a new address space containing the read-only mapped Zygote and a copy of the verified function code. By sharing the Zygote pages, multiple Trustlets can be spawned from the same Zygote while consuming only little additional memory and no additional initialization time. When a Trustlet attempts to write to a read-only Zygote memory page, a page fault handler performs the copy-on-write.

The Monitor allocates a new dedicated VMSA for the Trustlet. This VMSA is configured for execution in VMPL1 with user-level (ring 3) privileges. Additionally, the Monitor installs the page fault handler and activates the `Reflect #VC` feature to handle exceptions efficiently (detailed in Subsection 5.2.3).

Channel Setup for Function Chaining. If the new Trustlet is part of a function chain, the Monitor sets up channels to enable efficient communication between the chained Trustlets. These channels are shared memory regions mapped into the address spaces of connected Trustlets, allowing data to be passed without encryption and decryption overheads.

The Monitor creates an input channel for the first Trustlet and writes the input data to it. For each intermediate Trustlet in the chain, the Monitor creates a channel that connects its output to the next Trustlet's input. A final output channel is created for the last Trustlet, from which the Monitor will read the result.

5.2.3 Trustlet Execution

Once the Guest OS has set up the required Zygotes and Trustlets, CloakVM is ready for the function invocation itself. For function chains, the Monitor repeats the following process for every chain segment.

Trustlet Startup. The Monitor receives an encrypted payload, consisting of the function input and a symmetric key, from the Guest OS. The Monitor uses the private portion of the Function Key to decrypt these elements and places the function input in the Trustlet's input channel.

The Monitor proceeds to launch the Trustlet by invoking the hypervisor to run the vCPU in VMPL1 using the Trustlet's VMSA context (using the `SNP AP Creation` hypercall [8]).

Monitor Calls. Some system calls can be handled entirely within Gramine, while others require support from the CloakVM Monitor through the PAL. The Monitor needs to process certain calls directly (e. g., memory management operations) but safely delegates the others (e. g., file access requests) to the Guest OS.

- **Memory Management:** The Monitor handles requests for allocation, deallocation, protection changes, and memory-mapped file access. For instance, when a Trustlet requests memory allocation, the Monitor allocates a memory region, adjusts its access rights for VMPL1, and updates Trustlet's page table accordingly.

- **External File Access:** If a Trustlet requests to access a file that is not bundled within its Zygote image, the Monitor relays a message to the Guest OS to fetch the data. Once the Guest OS responds with the file contents, Gramine verifies its integrity (by checking its hash against the trusted file list in the manifest) before using it. While the Guest OS can observe the file that is accessed, it cannot alter it without detection.
- **Function Return:** When receiving a completion message from the function, the Monitor either launches the next Trustlet in the chain or, if it is the final Trustlet, continues with Subsection 5.2.4.

Trustlet–Monitor Interface. Normally, when a guest attempts an action that would require emulation from the hypervisor, the AMD SEV-SNP hardware raises a VMM Communication Exception (#VC) [27, 4], allowing the exception handler to decrypt the request into the GHCB before exiting the CVM. However, CloakVM instead uses the `Reflect #VC` feature [4] to redirect these exceptions from Trustlets (in VMPL1) to the Monitor (in VMPL0).[2]

To make a Monitor call, the Trustlet's LibOS places the request parameters into CPU registers (and a memory buffer for large data transfers). It then executes the `CPUID` instruction specifying a function number reserved for hypervisor use [5] as the parameter, which raises a #VC. The `Reflect #VC` feature causes the vCPU to exit so that the hypervisor can switch it to the higher-privileged VMPL0 VMSA, where the Monitor extracts the parameters from the Trustlet's VMSA and handles the request.

If the request must be forwarded to the Guest OS, the Monitor copies any necessary data from Trustlet's buffer to a Guest OS's buffer and lets the serverless orchestration framework handle the request. When the serverless orchestration framework is finished, it invokes the Trustlet again with the result, which the Monitor copies back to the Trustlet's memory.

Copy-on-Write Memory Management. If the Trustlet attempts to modify a memory page shared from the Zygote, the page fault event triggers a handler that calls the Monitor to allocate a private, writable copy of that page and replace the original page in the page table.

[2]`Reflect #VC` is designed primarily for unenlightened guests, which do not natively handle #VC events. In such cases, the SVSM must decrypt requests before (optionally) handing them off to the hypervisor. CloakVM uses this approach so that each Trustlet uses the Monitor's #VC handler instead of having to individually communicate with the hypervisor to change the execution to VMPL0.

5.2.4 Result Handling and Verification

When the Trustlet execution finishes — whether from a single Trustlet or the final one in a chain — CloakVM ensures the result's integrity and confidentiality before forwarding it to the serverless orchestration framework.

Result Encryption and Attestation Report Generation. The Monitor encrypts the function's output using the symmetric key that the caller provided.

The Monitor also compiles an attestation report that provides the measurements for the entire execution flow. Specifically, the attestation report includes:

- The AMD-SP attestation report that originally attested the Monitor itself.
- The cached measurements of the Zygote image and function code (multiple for chains).
- A measurement of the input.
- A measurement of the output.

The Monitor signs this report using the Function Key. The encrypted result and attestation report are returned to the Guest OS, where the serverless orchestration framework forwards them to the function caller.

Return to the Function Caller. The function caller verifies the signature of the attestation report and compares the attested input measurement with the expected one. If they match, the caller can be sure that the correct function was executed with the intended input in a secure environment. Otherwise, the Monitor would not have (if the Monitor was invalid) or not use (if it was provided with a different function than defined in the Policy) the private Function Key to sign the attestation report. If the hash of the output matches the one in the attestation report, the output has not been manipulated. The function caller can then decrypt the result using the symmetric key that it originally supplied.

This entire protocol has been formally verified to ensure secrecy, authenticity, and correctness [35].

6 Evaluation

6.1 Experimental Setup

To assess CloakVM, we compare it against other execution environments:

- **Native:** For our experiments, we use a server based on the AMD EPYC 7713P processor with 64 cores and 1 TB of RAM. It runs the Linux kernel in version 6.8 with AMD SEV-SNP and SVSM support [17]. For the native variant, we execute each function in a process. This approach lacks strong isolation and confidentiality guarantees, serving as a baseline for comparison.
- **Gramine [22]:** We assess the impact of executing workloads in a Gramine 1.7 environment to determine the performance cost associated with the LibOS approach.
- **Kata Containers [28]:** Kata Containers represents a lightweight VM-based approach that offers improved startup performance relative to traditional VMs while maintaining their isolation. We use QEMU [17] with Kernel-based Virtual Machine (KVM) as the hypervisor for this and all the following variants.
- **Virtual machine (VM):** We compare CloakVM to a standard VM running Ubuntu 22.04 with Linux 6.5, also with SVSM patches.
- **Confidential virtual machine (CVM):** To quantify the performance-confidentiality trade-off between a standard VM and a CVM, we also add AMD SEV-SNP CVMs to our comparison. The rest of the configuration is the same as for the VM case.
- **CloakVM:** Our new approach. We run CloakVM in a CVM with the same Guest OS as for the VM and CVM environment.

  CloakVM can be in one of three states when handling a request, which we evaluate separately:

  - **Cold start:** The Monitor is fully running, but the required Zygote and Trustlet have not been loaded yet.

- **Lukewarm start:** The required Zygote is loaded, but the Trustlet has not been created yet. This can be the case when another function using the same Zygote has been started before or when a new Trustlet has to be created due to user separation.
- **Warm start:** All necessary components are already loaded, and the function can start being executed immediately.

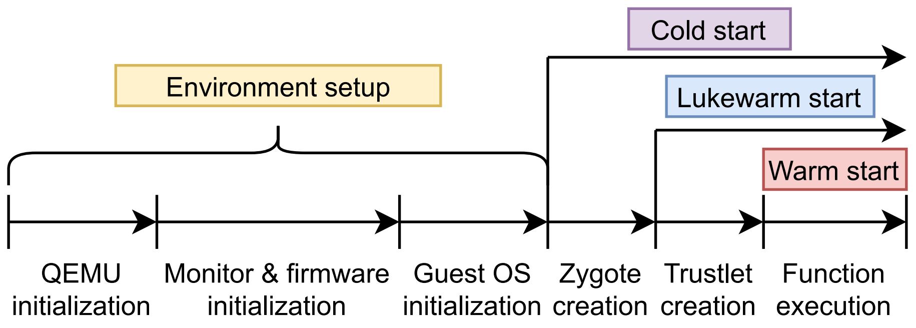

Figure 6.1: CloakVM invocation phases.

6.2 Microbenchmarks

We begin our evaluation by using microbenchmarks to quantify the specific performance aspects CloakVM aims to improve.

6.2.1 Startup Time

Many serverless functions are short-lived and latency-sensitive so that the startup overhead can become noticeable. We analyze the startup time by measuring how long it takes from starting a specific environment until the system can begin executing the function.

As we can see in Figure 6.2, booting a CVM for function execution takes a long time. In addition to a VM it has to perform a measurement at boot and validate all memory it uses, resulting in boot times approximately 2.2× slower than those of a standard VM. While CloakVM initialization takes roughly 1.5× the time of a CVM start, this process is a one-time cost. A CloakVM cold start is faster than a CVM and even a normal VM start. This is due to CloakVM running multiple functions in a single CVM. Kata Containers

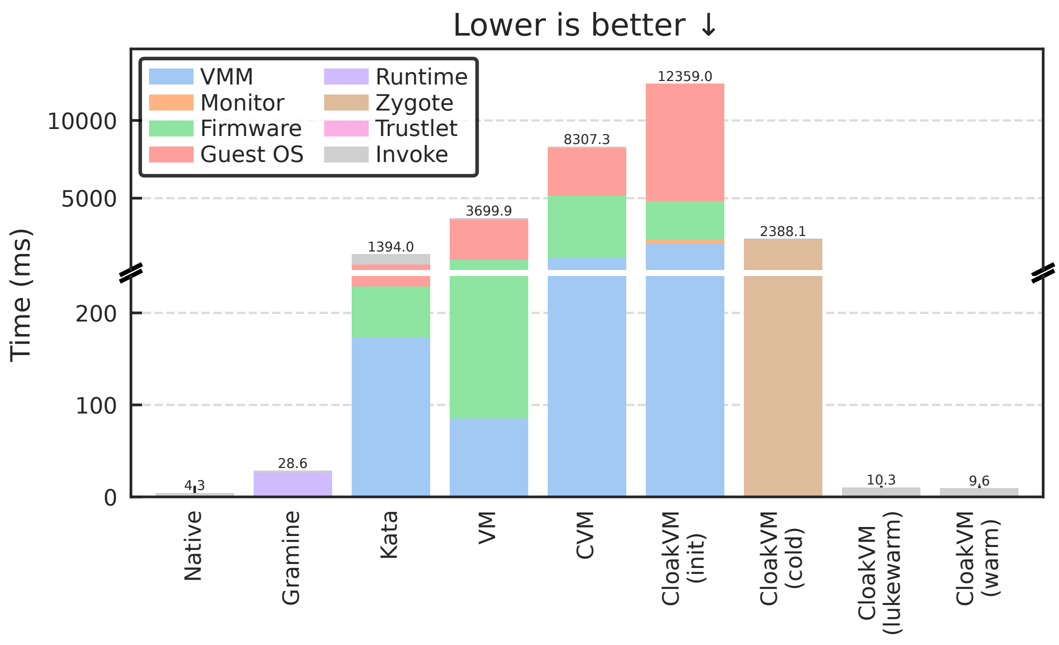

Figure 6.2: Startup time comparison [35].

is faster than a VM and faster than CloakVM at a cold start. However, lukewarm and warm starts of CloakVM are approximately two to three orders of magnitude faster and only about 2× native startup time. Native execution naturally offers the highest performance due to the absence of virtualization overhead.

This benchmark shows that adding confidentiality generally increases function startup time. However, thanks to CloakVM's Zygote optimization, lukewarm and warm starts have near-native startup time.

6.2.2 Attestation Cost

Although attestation is part of the overall startup process, we measure its cost separately to enable a clearer comparison with approaches that do not provide confidentiality and therefore skip attestation altogether.

To assess the impact of CloakVM's incremental attestation, we examine the time spent calculating measurements for the attestation report.

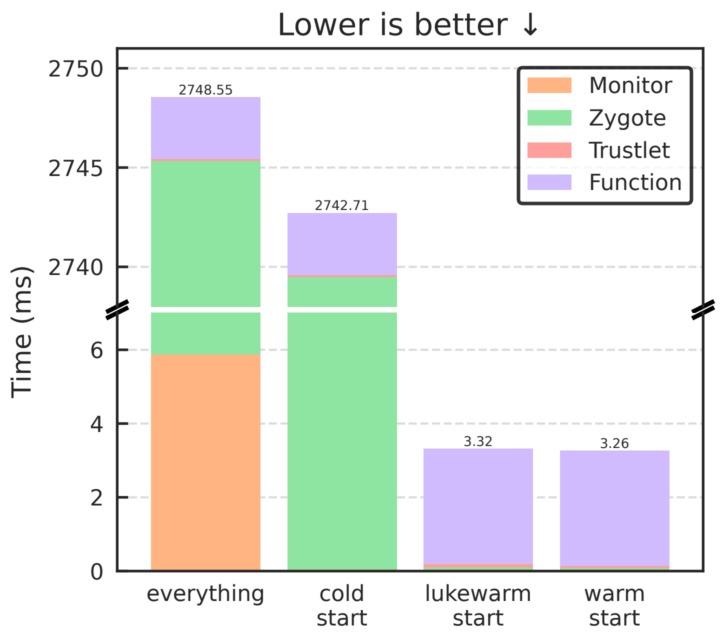

Figure 6.3: Measurement duration comparison [35].

Figure 6.3 shows that the Zygote measurement takes the longest of all components. Thanks to incremental attestation, this measurement only needs to happen in the cold start case and will be reused from then on. Compared to approaches that need to measure the whole environment at every start, CloakVM saves about 3 s at lukewarm and warm starts in this example case.

6.2.3 Communication Cost

To show our shared memory channels' impact on function chaining, we examine the time it takes to pass data for different payload sizes. For the comparison, we use a pipe for native, network communication for Gramine, Kata Containers, VM, CVM, and channels for CloakVM.

According to Figure 6.4, the CloakVM channels are very efficient, lying between Kata Containers and Gramine in performance. VMs and CVMs are significantly slower. CloakVM saves encrypting, decrypting, and copying back and forth via buffers compared to CVMs.

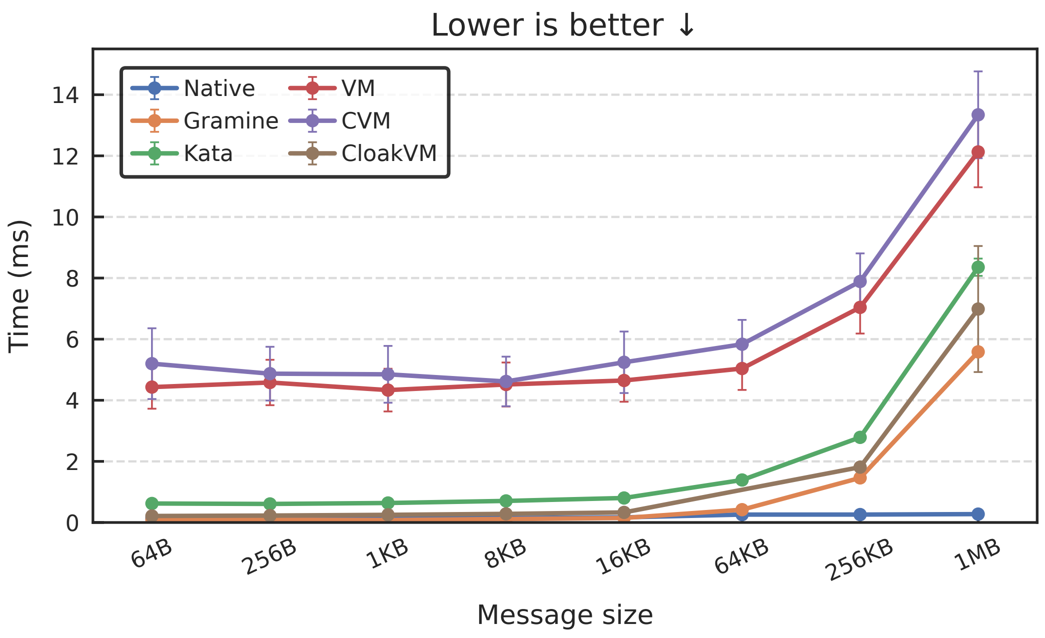

Figure 6.4: Communication time comparison [35].

6.3 TCB

Measuring the TCB is not straightforward. Table 6.1 provides a comparison of the TCB size in terms of lines of code (LOC), which serves as a rough estimate of the attack surface. However, LOC provide only limited insight into the actual security and complexity of the system. Important factors such as the formal verification of the communication protocol and the use of the Rust programming language for the Monitor are not accounted for by this metric.

Environment	**Host Kernel**	**Firmware/ Monitor**	**Guest Kernel**	**Runtime**	**Hypervisor**	**Total**
Gramine	1,115	903	–	36	–	2,054
Kata Containers	1,115	903	1,809	8,001	1,757	13,585
VM	1,115	903	3,177	744	1,757	7,696
CVM	–	903	3,177	744	–	4,824
CloakVM	–	332	–	780	–	1,112

Table 6.1: Trusted lines of code in thousands [35].

Nevertheless, we can observe that our approach has the lowest LOC of all the measured systems. This is the case because neither a hypervisor and host kernel (thanks to AMD SEV-SNP) nor a guest kernel (thanks to our Monitor) is included in our TCB.

6.4 Application-Level Benchmarks

We use the Python benchmarks from Serverless Benchmark Suite (SeBS) [18] to measure the performance of CloakVM in realistic scenarios. While the SeBS paper compares different cloud providers, we use its benchmarks to compare different execution environments.

SeBS's benchmarks consist of various classes of workloads to achieve high representativeness. We run the following benchmarks:[1]

- **thumbnailer:** thumbnail creation.
- **graph-mst:** graph minimum spanning tree construction.
- **dynamic-html:** dynamic HTML generation from a predefined template.
- **graph-pagerank:** graph PageRank calculation.
- **dna-visualisation:** visualization of DNA sequences.
- **compression:** compression of a set of files to an archive.
- **graph-bfs:** graph breadth-first search.

Compared to SeBS, we do not execute the "cost" and "container eviction" experiments because they are not applicable to CloakVM.[2] The "invocation overhead" experiment is unnecessary because we do not have to rely on a black-box approach and do measure it directly in Subsection 6.2.1.

We extend SeBS to support all our comparison environments and CloakVM. For the execution in CloakVM, we use the Python Zygote described in Subsection 5.1.1, modified to include the dependencies of the respective function in the embedded filesystem. We only include the files that are actually used.

[1]We skip the "uploader" and "video-processing" benchmarks since networking and child processes are currently not implemented in CloakVM.

[2]Cost is not applicable because the CloakVM project does not cover financials, and container eviction is not implemented in CloakVM at this time.

We measure the end-to-end latency: the duration from making a request to receiving the response. For the input size, we use the configuration option “small”. We execute the client and server on the same host with no network in between. Figure 6.5 shows the cold start comparison, which also includes lukewarm results, while Figure 6.6 presents the results for warm starts.

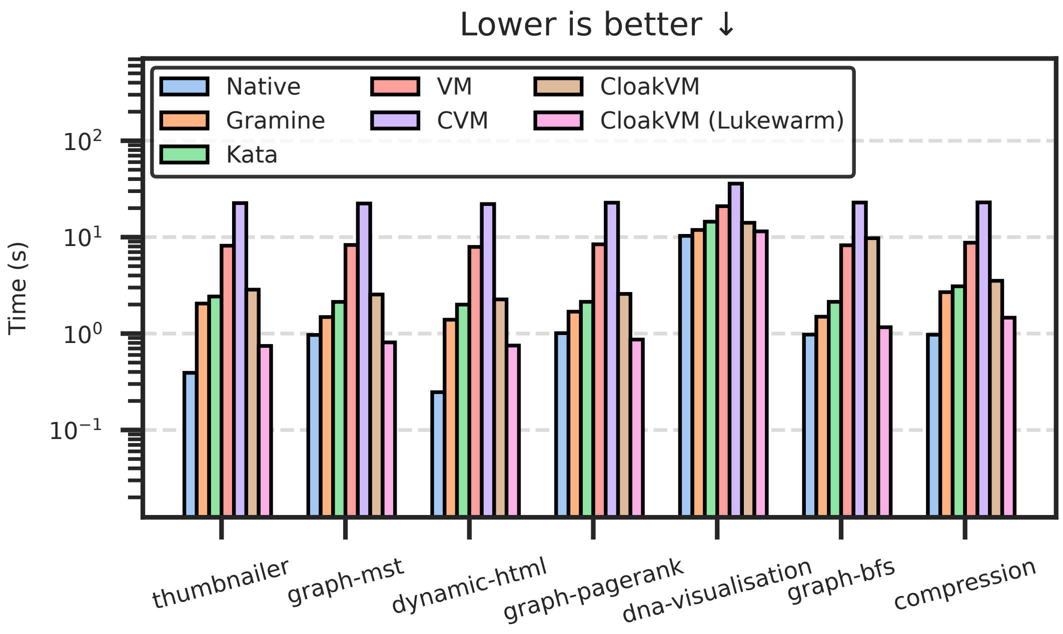

Figure 6.5: SeBS benchmarks: end-to-end latency for cold and lukewarm starts.

In general, CloakVM is slower compared to native, Gramine, and Kata Containers but faster than VM and CVM for cold starts and warm starts. On average, CloakVM is 83 % faster for cold starts and 13 % faster for warm starts compared to CVMs, for example.

CloakVM performs especially well during lukewarm starts as it is faster than Gramine and Kata Containers on cold starts. There, it is on average 94 % faster than CVMs.

Although CloakVM is slower than native, Gramine, and Kata Containers in many cases, it achieves stronger isolation through AMD SEV-SNP and Trustlet isolation.

Figure 6.7 shows that for all benchmarks, the Zygote creation is the phase that takes the most time. This shows why our Zygote mechanism (reusing pre-initialized processes) is so effective at reducing the startup time. Because each benchmark consists of relatively little code, the creation of Trustlets creation is fast. The input and output copying time depends on the data size but is still low compared to the Zygote creation time.

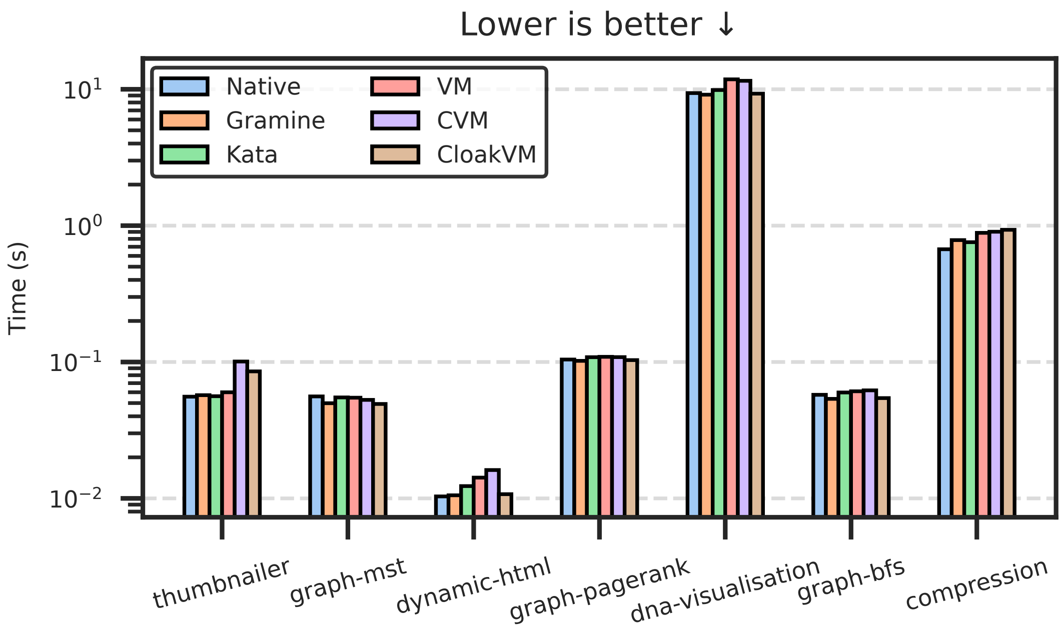

Figure 6.6: SeBS benchmarks: end-to-end latency for warm starts.

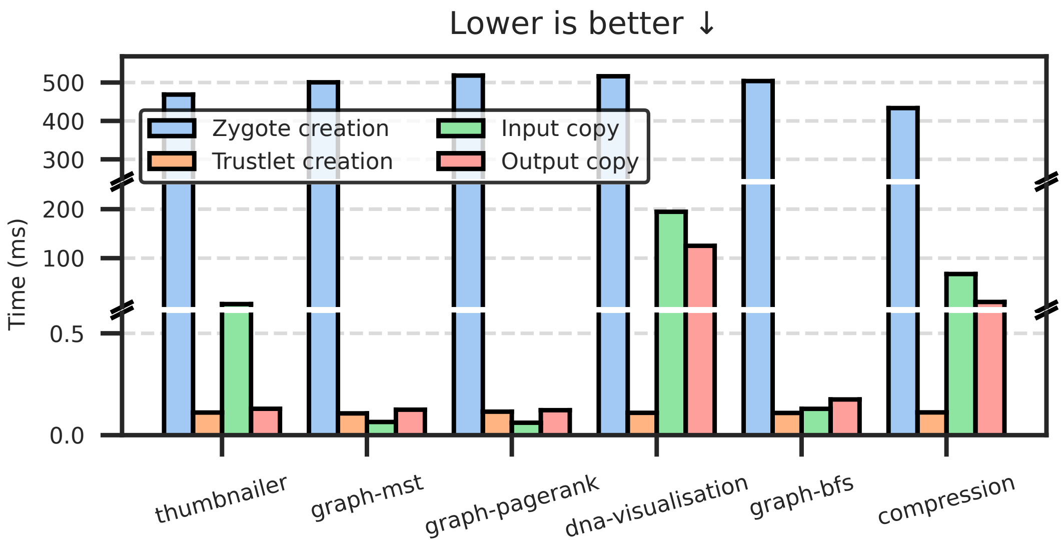

Figure 6.7: SeBS benchmarks: phase breakdown [35].

7 Related Work

Fast SGX-Based Confidential Serverless Platforms. Numerous systems leverage Intel SGX enclaves to provide confidentiality in serverless platforms. The following aim to address the associated performance overhead: Plug-In Enclaves [31] enable fast Intel SGX-based serverless functions by sharing memory between enclaves. Cryonics [29] reduces cold-start latency by snapshotting initialized enclaves and restoring them on demand.

CVM-Based Confidential Serverless Platforms. An alternative to enclaves is to secure serverless functions using CVMs. Serverless Confidential Containers [36] adopt this approach by running containerized workloads in CVMs. This design introduces significant cold-start latency, mainly due to memory provisioning and virtual firmware initialization.

Isolation with VMPLs. To address the overhead of starting and managing multiple CVMs, we look at intra-CVM isolation techniques using AMD SEV-SNP's VMPLs. Coconut SVSM [16] introduces a SVSM that provides secure services within a CVM, though it does not support executing applications. Veil [9] protects applications by providing secure services and enclaves, similarly to our Trustlets. NestedSGX [41] uses VMPLs to enable Intel SGX enclaves to run within a CVM.

LibOSes for TEEs. LibOSes allow executing unmodified Linux applications in small TEEs without relying on a traditional OS. Gramine-SGX [40] allows unmodified applications to run inside Intel SGX enclaves. Gramine-TDX [30] extends this capability to Intel TDX CVMs.

CloakVM builds on the VMPL-based isolation work and applies it to the serverless domain. It supports unmodified Linux applications through Gramine. CloakVM uses memory-sharing for its Zygotes, enabling both high performance and strong confidentiality guarantees for serverless functions.

8 Conclusion and Future Work

This thesis presents CloakVM, a confidential computing system designed to deliver both strong security and high performance for confidential serverless functions by leveraging AMD SEV-SNP's hardware-enforced isolation, implementing a small, trusted Monitor, and utilizing Zygotes and memory channels.

CloakVM is built on AMD SEV-SNP and uses its trusted execution environment (TEE) as the foundation for its isolation. Having a small Monitor and not trusting the Guest OS results in a small TCB. The design of the attestation protocol enables incremental and local verification of functions, reducing latency without sacrificing security. The copy-on-write Zygote mechanism allows efficient memory utilization and fast startups while maintaining isolation between functions. And the memory channels between Trustlets in a function chain eliminate the overhead of re-encryption and re-attestation at function boundaries.

Everything together makes CloakVM well-suited for security-sensitive serverless workloads in untrusted cloud environments.

Several ideas for future work emerge from this project:

- Confidential Function Code
  Currently, function code is loaded in plaintext from the registry and only the function input is encrypted by the function caller. In scenarios where the function logic contains sensitive information (e. g., intellectual property or proprietary algorithms), it could also be encrypted, ensuring only the CloakVM Monitor can decrypt it.
- Input Integrity and Replay Protection
  While CloakVM encrypts input data, it does not yet enforce integrity protection. As a result, the untrusted Guest OS could theoretically replay valid requests. This might be harmless. However, certain functions require to be called only once. Future work could research how to guarantee the input's authenticity and prevent replay.

- Function Signing
  At present, the Monitor stores a hash of each function in its Policy. If a function needs to be updated, its hash changes, requiring a Policy update. A more flexible approach involves code signing: the function provider signs the function code, and the Policy stores only the provider's public key rather than a hash. As long as the signature is valid, the updated function code is recognized as legitimate. This avoids repeated Policy updates and simplifies continuous deployment.

- Networking Support
  CloakVM already delegates file-system operations to the untrusted Guest OS. This mechanism could be extended to handle network requests in a similar manner.

- Multiprocessing
  Each Trustlet currently operates as a single process. Some workloads are CPU-intensive and benefit from parallelization across multiple cores. Future work could enable multiprocessing inside a Trustlet.

- Integration With Serverless Orchestration Frameworks
  CloakVM includes a small Python-based orchestration library for demonstration. A potential next step for this project could be integration with existing frameworks such as Apache OpenWhisk [14] or Knative [38]. This integration would automatically route function triggers, handle scaling, and manage complex workflows with the confidentiality and performance benefits of CloakVM.

Appendix

This appendix provides an overview of the source code developed or used throughout this research. It resides in the following GitHub repositories:

- COCONUT-SVSM: `https://github.com/coconut-svsm/svsm`.
- Gramine: `https://github.com/gramineproject/gramine`.
- Serverless Benchmark Suite (SeBS) fork for CloakVM: `https://github.com/TUM-DSE/SeBS/tree/michael`.
- SeBS fork for the baselines: `https://github.com/TUM-DSE/SeBS/tree/normal-benchmarks`.
- SeBS data: `https://github.com/TUM-DSE/serverless-benchmarks-data`.
- Benchmark scripts: `https://github.com/TUM-DSE/CVM_eval/tree/sebs/benchmarks/sebs`.

Abbreviations

AMD-SP	AMD Secure Processor
CA	certificate authority
CVM	confidential virtual machine
DH	Diffie-Hellman
FaaS	Function as a Service
GHCB	Guest-Hypervisor Communication Block
GPA	guest physical address
KVM	Kernel-based Virtual Machine
LibOS	library operating system
LOC	lines of code
PAL	Platform Adaptation Layer
RMP	Reverse Map Table
SeBS	Serverless Benchmark Suite
SEV-SNP	Secure Encrypted Virtualization with Secure Nested Paging

SGX	Software Guard Extensions
SPA	system physical address
SVSM	Secure VM Service Module
TCB	trusted computing base
TDX	Trust Domain Extensions
TEE	trusted execution environment
#VC	VMM Communication Exception
vCPU	virtual central processing unit
VM	virtual machine
VMPL	Virtual Machine Privilege Level
VMSA	Virtual Machine Save Area
vTPM	virtual Trusted Platform Module

List of Figures

List of Tables

Bibliography

[1] Advanced Micro Devices. *Advanced Micro Devices Versioned Chip Endorsement Key (VCEK) Certificate and KDS Interface Specification*. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/57230.pdf. Version 1.00. 2025.

[2] Advanced Micro Devices. *AMD Secure Encrypted Virtualization (SEV)*. URL: https://www.amd.com/en/developer/sev.html (visited on 04/04/2025).

[3] Advanced Micro Devices. *AMD 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-strengthening-vm-isolation-with-integrity-protection-and-more.pdf. 2020.

[4] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2*. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf. Version 3.42. 2024.

[5] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 3*. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24594.pdf. Version 3.36. 2024.

[6] Advanced Micro Devices. *Secure VM Service Module for SEV-SNP Guests*. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/58019.pdf. Version 1.00. 2023.

[7] Advanced Micro Devices. *SEV Secure Nested Paging Firmware ABI Specification*. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56860.pdf. Version 1.57. 2025.

[8] Advanced Micro Devices. *SEV-ES Guest-Hypervisor Communication Block Standardization*. https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56421.pdf. Version 2.04. 2025.

[9] A. Ahmad, B. Ou, C. Liu, X. Zhang, and P. Fonseca. "Veil: A Protected Services Framework for Confidential Virtual Machines." In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. ASPLOS '23. Vancouver, BC, Canada: Association for Computing Machinery, 2024, pp. 378–393. ISBN: 9798400703942. DOI: `10.1145/3623278.3624763`.

[10] Amazon Web Services. *Lambda executions*. URL: `https://docs.aws.amazon.com/whitepapers/latest/security-overview-aws-lambda/lambda-executions.html#lambda-microvms-and-workers` (visited on 03/12/2025).

[11] Amazon Web Servies. *Amazon Web Services (AWS)*. URL: `https://aws.amazon.com/` (visited on 04/04/2025).

[12] Amazon Web Servies. *AMD SEV-SNP for Amazon EC2 instances*. URL: `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/sev-snp.html` (visited on 04/14/2025).

[13] Amazon Web Servies. *AWS Lambda*. URL: `https://aws.amazon.com/lambda/` (visited on 04/04/2025).

[14] Apache OpenWhisk. *Open Source Serverless Cloud Platform*. URL: `https://openwhisk.apache.org/` (visited on 04/04/2025).

[15] E. Brewer. *gVisor: Protecting GKE and serverless users in the real world*. URL: `https://cloud.google.com/blog/products/containers-kubernetes/how-gvisor-protects-google-cloud-services-from-cve-2020-14386?hl=en` (visited on 03/12/2025).

[16] coconut-svsm. *COCONUT Secure VM Service Module*. URL: `https://coconut-svsm.github.io/svsm/` (visited on 04/04/2025).

[17] coconut-svsm. *Installing the COCONUT-SVSM*. URL: `https://coconut-svsm.github.io/svsm/installation/INSTALL/` (visited on 04/04/2025).

[18] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler. "SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing." In: *Proceedings of the 22nd International Middleware Conference*. Middleware '21. Québec city, Canada: Association for Computing Machinery, 2021, pp. 64–78. ISBN: 9781450385343. DOI: `10.1145/3464298.3476133`.

[19] Google. *Cloud Run functions*. URL: `https://cloud.google.com/functions` (visited on 04/04/2025).

[20] Google. *Confidential VM overview*. URL: `https://cloud.google.com/confidential-computing/confidential-vm/docs/confidential-vm-overview` (visited on 04/14/2025).

[21] Google. *Google Cloud*. URL: https://cloud.google.com/ (visited on 04/04/2025).

[22] Gramine Contributors. *Gramine documentation*. URL: https://gramine.readthedocs.io/ (visited on 04/04/2025).

[23] Gramine Contributors. *Gramine features*. URL: https://gramine.readthedocs.io/en/stable/devel/features.html (visited on 04/04/2025).

[24] Intel Corporation. *Intel® Software Guard Extensions*. URL: https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html (visited on 03/13/2025).

[25] Intel Corporation. *Intel® Trust Domain Extensions (Intel® TDX)*. URL: https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html (visited on 03/13/2025).

[26] D. Kaplan. "Hardware VM Isolation in the Cloud: Enabling confidential computing with AMD SEV-SNP technology." In: *Queue* 21.4 (Sept. 2023), pp. 49–67. ISSN: 1542-7730. DOI: 10.1145/3623392.

[27] D. 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. 2017.

[28] Kata Containers. *Kata Containers*. URL: https://katacontainers.io/ (visited on 04/04/2025).

[29] S.-J. Kim, M. You, B. J. Kim, and S. Shin. "Cryonics: Trustworthy Function-as-a-Service using Snapshot-based Enclaves." In: *Proceedings of the 2023 ACM Symposium on Cloud Computing*. SoCC '23. Santa Cruz, CA, USA: Association for Computing Machinery, 2023, pp. 528–543. ISBN: 9798400703874. DOI: 10.1145/3620678.3624789.

[30] D. Kuvaiskii, D. Stavrakakis, K. Qin, C. Xing, P. Bhatotia, and M. Vij. "Gramine-TDX: A Lightweight OS Kernel for Confidential VMs." In: *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. CCS '24. Salt Lake City, UT, USA: Association for Computing Machinery, 2024, pp. 4598–4612. ISBN: 9798400706363. DOI: 10.1145/3658644.3690323.

[31] M. Li, Y. Xia, and H. Chen. "Confidential Serverless Made Efficient with Plug-In Enclaves." In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021, pp. 306–318. DOI: 10.1109/ISCA52012.2021.00032.

[32] Microsoft. *About Azure confidential VMs*. URL: https://learn.microsoft.com/en-us/azure/confidential-computing/confidential-vm-overview (visited on 04/14/2025).

[33] Microsoft. *Azure Functions*. URL: `https://azure.microsoft.com/en-us/products/functions` (visited on 04/04/2025).

[34] Microsoft. *Microsoft Azure*. URL: `https://azure.microsoft.com/en-us/` (visited on 04/04/2025).

[35] P. Sabanic, M. Hackl, T. Bodea, P. Julian, M. Misono, D. Stavrakakis, and P. Bhatotia.

[36] C. Segarra, T. Feldman-Fitzthum, D. Buono, and P. Pietzuch. "Serverless Confidential Containers: Challenges and Opportunities." In: *Proceedings of the 2nd Workshop on SErverless Systems, Applications and MEthodologies*. SESAME '24. Athens, Greece: Association for Computing Machinery, 2024, pp. 32–40. ISBN: 9798400705458. DOI: `10.1145/3642977.3652097`.

[37] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. "Serverless in the wild: characterizing and optimizing the serverless workload at a large cloud provider." In: *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC'20. USA: USENIX Association, 2020. ISBN: 978-1-939133-14-4.

[38] The Knative Authors. *Knative*. URL: `https://knative.dev/docs/` (visited on 04/04/2025).

[39] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter. "Cooperation and security isolation of library OSes for multi-process applications." In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: Association for Computing Machinery, 2014. ISBN: 9781450327046. DOI: `10.1145/2592798.2592812`.

[40] C.-C. Tsai, D. E. Porter, and M. Vij. "Graphene-SGX: a practical library OS for unmodified applications on SGX." In: *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '17. Santa Clara, CA, USA: USENIX Association, 2017, pp. 645–658. ISBN: 9781931971386.

[41] W. Wang, L. Song, B. Mei, S. Liu, S. Zhao, S. Yan, X. Wang, D. Meng, and R. Hou. "The Road to Trust: Building Enclaves within Confidential VMs." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2025. DOI: `10.14722/ndss.2025.240385`.