WALLET: Confidential Serverless Computing

Patrick Sabanic Masanori Misono Teofil Bodea Julian Pritzi
Michael Hackl Dimitrios Stavrakakis Pramod Bhatotia
Technical University of Munich

Abstract

Although serverless computing offers compelling cost and deployment simplicity advantages, a significant challenge remains in securely managing sensitive data as it flows through *the network of ephemeral function executions* in serverless computing environments within untrusted clouds. While Confidential Virtual Machines (CVMs) offer a promising secure execution environment, their integration with serverless architectures currently faces fundamental limitations in key areas: *security*, *performance*, and *resource efficiency*.

We present WALLET, a lightweight confidential computing system for secure serverless deployments. By employing nested confidential execution and a decoupled guest OS within CVMs, WALLET runs each function in a minimal "trustlet", significantly improving security through a reduced Trusted Computing Base (TCB). Furthermore, by leveraging a data-centric I/O architecture built upon a lightweight LibOS, WALLET optimizes network communication to address performance and resource efficiency challenges.

Our evaluation shows that compared to CVM-based deployments, WALLET has a 4.3× smaller TCB, improves end-to-end latency (15–93%), achieves higher function density (up to 907×), and reduces inter-function communication (up to 27×) and function chaining latency (16.7-30.2×); thus, WALLET offers a practical system design for confidential serverless computing.

1 Introduction

Serverless computing is a rapidly evolving cloud paradigm offering low costs and simplified deployment [70]. As serverless applications increasingly manage sensitive data, particularly in AI/ML, strong security measures are of utmost importance [19, 22]. However, securely managing data within serverless computing is particularly challenging due to their ephemeral and distributed execution, where short-lived functions are chained through network communication [8, 79, 152].

To this end, Confidential Virtual Machines (CVMs) [16, 17, 77] provide a promising approach for the secure execution of serverless workloads in untrusted cloud environments [33], as they protect the confidentiality and integrity of entire VMs and do not require significant application modifications [18, 76].

However, due to strict serverless computing requirements, simply deploying serverless workloads directly within CVMs reveals critical inherent limitations (§ 3). In particular, despite its portability benefits, this approach exhibits *non-ideal security properties*, incurs *prohibitive performance overheads*, and leads to *costly resource inefficiencies*. In terms of security, CVMs rely on a full guest OS to host serverless applications, resulting in a bloated Trusted Computing Base (TCB) that expands the attack surface due to potential vulnerabilities in the untrusted OS (§ 3.1). From a performance perspective, CVMs suffer from long boot times—often exceeding function execution duration—due to strict security mechanisms (e.g., memory encryption) (§ 3.2). They also introduce significant I/O communication overheads, which are particularly detrimental in function chaining scenarios (§ 3.3) and require costly per-instance attestation, adding considerable latency to short-lived executions (§ 3.6). Finally, resource efficiency is severely impacted as CVM-based serverless frameworks struggle with function scheduling (§ 3.4), primarily due to consolidation challenges stemming from ineffective memory deduplication and hardware density constraints (§ 3.5).

Despite the advancements in virtualization architectures, *no existing approach can fully resolve the combined challenges of security, performance, and resource efficiency in confidential serverless computing*. Traditional VMs provide hardware-based isolation but incur substantial boot delays [103], high inter-VM communication costs [156], and maintain large TCBs because of their full virtualization stacks. Lightweight VMs (e.g., FireCracker [3], Kata [42]) optimize function density and startup latency [102], and reduce TCBs by trimming hypervisor and guest OS components [3]; however, they still suffer from significant I/O overheads and designs that complicate security verification [41, 140, 148]. Specialized OS architectures (e.g., LibOSes [27, 88, 136], Unikernels [87, 100, 111]) deliver near-instant startups by compiling applications with minimal kernel libraries but require developer efforts to craft application-specific images and sacrifice compatibility. CVMs (e.g., AMD SEV-SNP [9], Intel TDX [77], ARM CCA [17]), in turn, provide hardware-based security properties but introduce significant network communication overheads [92, 112, 130]. In summary, while each approach addresses individual aspects of the problem, a holistic solution that reconciles all these concerns remains an open challenge.

As a solution, we propose WALLET, a lightweight confidential computing system that enables secure serverless deployments in untrusted clouds by fundamentally rethinking how sensitive workloads are isolated and executed inside CVMs. In WALLET, each function runs in its own *trustlet*—a minimal serverless process that executes within a secure environment inside a CVM. With this design, client requests are securely routed through the untrusted guest OS, while the actual ex-

ecution occurs in trustlets that encapsulate only the critical application code and the minimal system support required to serve these functions. This arrangement leverages two key mechanisms: *nested confidential execution* and a *decoupled guest OS architecture*. Inspired by nested virtualization [24] and ARM's TrustZone model [18, 133], we partition the CVM space into smaller nested isolated regions. By confining sensitive code to trustlets, Wallet significantly reduces the overall TCB and minimizes the attack surface, thereby overcoming the limitations of hosting workloads on a monolithic guest OS.

To achieve high performance and fast function instantiation, Wallet adopts several techniques. First, it employs pre-initialized *process templates* (inspired by Android's zygote model [37, 89]) to expedite the bootstrapping of trustlets. Complementing this, a *dynamically loadable LibOS* enables trustlets to achieve the minimal startup latency characteristic of unikernel-based systems [100, 124, 166] while retaining flexibility for diverse workloads. Moreover, Wallet introduces a *data-centric I/O architecture* that leverages high-density function co-location and replaces conventional CVM networking for inter-function communication, optimizing communication paths via secure *data objects*. It also incorporates *differential attestation*-an innovative technique that reuses pre-measured components to reduce attestation latency by focusing on only the mutable parts during invocation.

Collectively, these mechanisms enhance resource efficiency. As process templates and the LibOS are designed to share memory, and trustlets are ephemeral—occupying resources only during function execution—Wallet achieves a higher density of function deployments per CVM. This tight integration of performance optimizations with a thin TCB not only accelerates startup and execution but also allows for more efficient resource utilization, thereby supporting a larger number of concurrent function requests within the same memory footprint.

We implement Wallet on AMD SEV-SNP [9] with source code and evaluation setup to be **publicly available**. Our evaluation shows Wallet reduces TCB by 4.3× compared to CVMs while achieving lower end-to-end latency in real serverless applications for both cold (85.10-93.44%) and warm (14.97%) starts. In large-scale simulations using Azure Functions traces [142, 165], Wallet achieves significantly lower invocation latency than CVMs (i.e., 5ms vs. 489s at the 50th percentile, 1.5s vs. 881s at the 99th percentile). Further, performance microbenchmarks indicate that Wallet's cold starts are 3.5× faster than CVMs and 35% than traditional VMs, while warm starts show negligible latency (<10.3 ms). Function communication latency decreases by 1.4−27× thanks to Wallet's data-centric networking architecture, while function chaining performance improves by 16.7−30.2× over CVMs. Attestation latency is also reduced as Wallet minimizes the measurement time (0.25 ms in warm starts). Lastly, Wallet has a smaller memory footprint than CVMs (up to 907×) for the same number of concurrent functions, rendering Wallet an ideal solution for secure serverless deployments in untrusted clouds.

2 Confidential Virtual Machines (CVMs)

As an alternative to existing cloud serverless virtualization architectures, CVMs [59, 69, 107] (e.g., AMD SEV-SNP [9], Intel TDX [77], ARM CCA [17]) provide a promising way to address escalating security requirements while preserving compatibility [112]. Precisely, CVMs protect the VM code, data, and memory from unauthorized access, including the hypervisor, via hardware-enforced isolation and encryption.

During a CVM launch, the VMM loads initial guest code, data, and state but cannot access the guest memory or modify its state, beyond this point. Guest firmware configures the private memory and performs necessary measurements. The guest kernel must be CVM-aware to interact with the trusted hardware. Host communication uses controlled, encrypted hypercalls. CVMs support remote attestation to verify their initial state and platform integrity. While providing strong internal security, they require secure protocols (e.g., networking) for data in transit/at rest to maintain their security guarantees.

Currently, Linux supports Intel TDX, AMD SEV-SNP, and ARM CCA (the hardware support of ARM CCA is absent as of now). In addition, Gramine-TDX [88] minimizes the guest TCB by employing a LibOS architecture and specializing the OS to a specific application on the Intel TDX platform.

CVM networking architecture. Since the hypervisor cannot directly access the CVM's memory, CVMs use an unencrypted shared buffer (*bounce buffer* [130]) for network communication and employ encrypted protocols (e.g., TLS). Specifically, typical CVM networking involves [92]: *(i)* guest kernel encrypts and copies data to a bounce buffer, *(ii)* hypervisor transfers it to the destination CVM's bounce buffer, and *(iii)* the destination CVM copies and decrypts the data for future consumption. This pipeline causes delays due to multiple copies, cryptographic operations, and heavy context switches [112, 158].

CVM partitioning. To improve intra-CVM isolation, CVMs offer a layered privilege model [1, 72, 110] beyond ring protections. This partitioning assigns each vCPU a privilege level defining its memory access rights and allowed instructions. The highest privilege level can intercept events from lower levels, enabling the provision of secure services (e.g., vTPM [71, 114]) without hypervisor dependence, and potentially shrinking the TCB by trusting only the most privileged components.

Specifically, AMD SEV-SNP's Virtual Machine Privilege Levels (VMPL) [10] offer four levels (VMPL-0 to VMPL-3), with VMPL-0 being the most privileged. Higher levels can modify lower-level permissions. Intel's TD-partitioning [72] enables complete nested virtualization within a CVM, allowing up to three nested VMs with an L1 VMM fully controlling the guests.

3 Motivation

Confidential computing [33] offers a promising way to secure serverless workloads in the untrusted cloud. However, CVM-based serverless computing faces *six* key challenges due to mismatches between CVM constraints and serverless requirements, analyzed below (experimental setup is described in § 7).

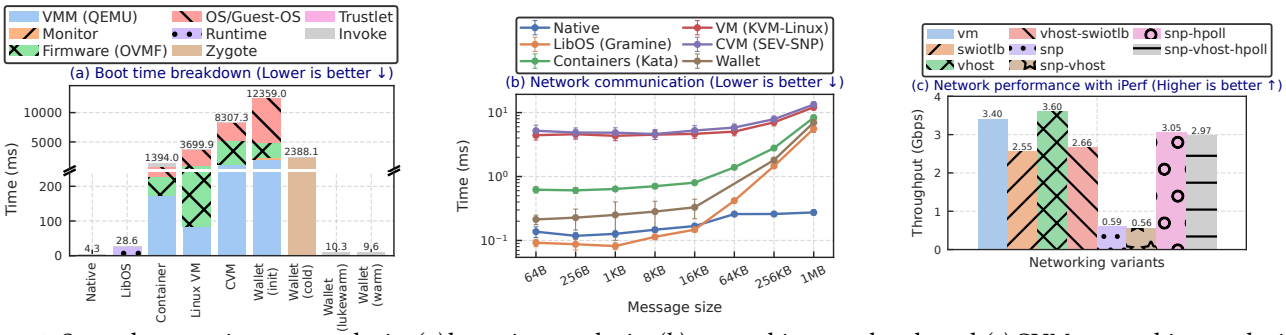

Figure 1: Serverless requirement analysis: *(a)* boot time analysis, *(b)* networking overheads and *(c)* CVM networking analysis.

Table 1: TCB size comparison. Presented values are KLoC.

Baseline	Host Kernel	Firm-ware	Guest Kernel	Runtime	Hyper-visor	Total
LibOS (Gramine)	1,115	903	—	36	—	2,054
Containers (Kata)	1,115	903	1,809	8,001	1,757	13,585
VM (KVM-Linux)	1,115	903	3,177	744	1,757	7,696
CVM (SEV-SNP)	—	903	3,177	744	—	4,824
WALLET	—	332	—	780	—	1,112

3.1 Problem #1: Large Trusted Computing Base

Minimizing the TCB is crucial for reducing the attack surface in confidential serverless computing. While CVMs eliminate hypervisor dependencies, CVM-hosted serverless stacks retain large TCBs containing monolithic legacy software with redundant components for backward compatibility [88].

Table 1 showcases that the CVM TCB size (4.8M LoC) is smaller than traditional VMs (7.7M LoC) but much larger than LibOS alternatives (e.g., Gramine LibOS: 2M LoC). Thus, CVM-hosted serverless stacks do not adequately minimize the TCB. The inclusion of a full guest OS increases complexity and exposes a large attack surface. In contrast, WALLET achieves approximately 4× TCB reduction compared to CVMs and 2× compared to Gramine LibOS without compromising security.

3.2 Problem #2: Slow Boot Times

Low startup latency is vital for serverless functions, but CVMs introduce inherent boot penalties [65, 112] due to three security mechanisms [14, 74]: *(i)* costly per-guest key generation and CVM memory pre-encryption, *(ii)* explicit guest state/memory initialization and boot measurement calculation for attestation, and *(iii)* synchronous signed report generation and external validation delaying function execution.

To this end, we measure the boot time of different virtualization architectures (Figure 1a), which indicate that CVMs exhibit a boot time of ~8.3 seconds, 2.2× slower than standard VMs (3.7s), primarily due to VMM setup, OS initialization, and encrypted memory configuration. These inevitable security-induced boot delays often exceed function lifetimes, making CVMs impractical for latency-sensitive serverless computing. While WALLET has overall the highest initialization time (~ 12s), it is an one-time cost. In contrast, with an initialized WALLET, we can achieve faster cold starts (2.4 s) than traditional VMs (3.7 s), and a near-negligible warm start latency (<10.3 ms). Notably, this warm start achieves even lower start-up times compared to the LibOS (~ 28 ms) on its own running without isolation on the host, thanks to its efficient CoW-based function execution.

3.3 Problem #3: High Networking Overheads

High-performance networking is crucial in distributed serverless applications with frequent data exchanges (e.g., function chaining). However, CVMs introduce significant networking overheads due to strict isolation and the requirement for additional encryption (e.g., TLS) to secure data in transit, which involves heavy context switches [30, 112].

To gauge these overheads, we measure the communication costs between two instances of the same virtualization model with varying message sizes (Figure 1b). We observe that CVMs consistently exhibit higher latency compared to traditional VMs (9-20%) and lightweight solutions such as Kata containers and Gramine (up to 59×).

We further examine the network performance of CVMs with various Linux configurations. Figure 1c shows iPerf [78] throughput (UDP, 1460 bytes payload, client runs on the host). "swotlb" means that bounce buffer is enabled, and "vhost" implies using the vhost optimization. The SNP VM *always* uses a bounce buffer. We observe that the SNP VM suffers from performance degradation (only 17% of the VM), and the guest-side halt polling [46] ("-hpoll") mitigates the performance overhead at the cost of excessive CPU usage [112]. In contrast, WALLET achieves lower latency than CVMs (1.4-27×) leveraging function co-location and following a data-centric networking model for secure, yet performant, communication.

3.4 Problem #4: Inefficient Scheduling

Efficient scheduling is a critical challenge [35, 98, 168] in confidential serverless computing. While optimized two-level (global/local) schedulers [64, 144, 161] favor warm starts, the limited CVMs per node [15, 73, 75, 125] can lead to more cold starts, increasing invocation latency. Further, traditional local schedulers, such as the Completely Fair Scheduler (CFS), are also ineffective [168] for short functions [67].

Specifically, confidential serverless functions in isolated environments cannot share resources, restricting the flexibility of resource-aware schedulers [144]. The limited pool of instances per CVM node forces balancing instance reuse,

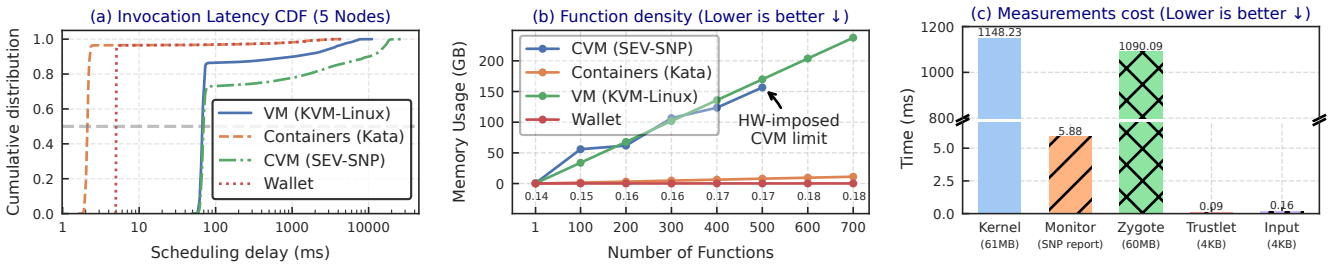

Figure 2: Serverless requirement analysis: *(a)* the scheduling delay, *(b)* the function density, and *(c)* attestation measurements.

increasing cold start ratios [98, 168]. In addition, switching execution context between CVMs incurs significant overheads (e.g., TLB/cache flushing, memory validation) [28, 112, 169]. Our large-scale system simulation using sampled [154] Azure Functions traces [142, 165] (30 minutes, 4,000 functions, 4.1 million invocations) shows CVMs having higher scheduling latency across all percentiles compared to VMs and microVMs (Figure 2a) under the same scheduling model.

Thus, we conclude that standard scheduling is suboptimal for CVM-based serverless. Wallet achieves lower scheduling latency with better tail latencies by sustaining more warm instances per node and using a run-to-completion model [168].

3.5 Problem #5: Impractical Server Consolidation

Efficiently packing multiple functions onto shared hardware resources is vital in serverless computing [3, 95], but also challenging for monolithic CVM-based serverless deployments for three fundamental reasons. First, CVM's encrypted memory makes effective memory deduplication difficult, if not technically impossible [15, 73, 75, 126], as each memory is encrypted with a different key. Second, hardware limits the number of CVMs per node [15, 73, 75, 125], creating a hard ceiling on function density. Lastly, CVMs' heavy-weight stacks cause higher per-function memory overhead, reducing achievable density considering a fixed hardware budget.

Figure 2b presents the memory usage of concurrent functions in a node. CVMs have the steepest increase, consuming ~168 GB for 500 functions. The respective number for MicroVMs is ~8.5 GB. Figure 2b further highlights the limit of CVMs per node imposed by the hardware (~500 in our case).

Thus, CVM-based serverless struggles to consolidate functions efficiently. In contrast, Wallet can host multiple isolated functions within a CVM (up to 907× less memory than CVMs), exceeding the hardware-imposed density limit.

3.6 Problem #6: High Attestation Overheads

Remote attestation [106] establishes trust in a CVM, but its second-scale latency per CVM is excessive for ms-scale serverless functions. Specifically, attestation involves measuring software components and generating a signed report. To ensure function execution integrity, additional measurements of the function and input/output are necessary. This process can become costly, especially in function chaining scenarios.

Figure 2c shows that the measurement of a full guest OS kernel for a Linux CVM can take up to 1.1 s, significantly increasing the overall startup latency. While long-running workloads can amortize this cost, the short execution times of serverless functions make the repetitive attestation overhead prohibitive. Our Wallet prototype significantly reduces the measurement cost to ~0.25 ms for warm starts by reusing already calculated measurements across the same functions.

3.7 Summary and Problem Statement

CVMs face critical limitations when employed in serverless settings. Their large TCB, slow boot times, I/O communication overhead, and costly attestation render them ill-suited for serverless computing. These constraints further lead to inefficient scheduling and low function density due to hardware limitations and suboptimal resource management.

To this end, there exists prior work that focuses on providing specialized solutions to tackle these challenges [65, 88]. Specifically, Gramine-TDX [88] provides a slim, security-first OS kernel, based on Gramine LibOS [153], to run unmodified applications in CVMs while having a minimal TCB. However, it is mainly suited for single-tenant architectures, where each serverless function requires its dedicated CVM, which conflicts with confidential serverless computing requirements.

Precisely, Gramine-TDX cannot achieve efficient function consolidation due to hardware CVM limits, requires per-function attestation, and forces inter-function communication through expensive CVM networking. Further, Gramine-TDX cannot leverage function co-location opportunities within the same CVM in a secure manner, as it operates with system-wide privileges and lacks the fine-grained privilege separation mechanisms necessary to safely isolate multiple co-located functions from each other and from the underlying software layers. These architectural constraints make Gramine-TDX unsuitable for the multi-tenant, high-density, and communication-intensive nature of serverless workloads.

Problem statement. To materialize confidential serverless computing, we have to answer a critical question: *how can we execute lightweight, short-lived functions in the untrusted cloud in a secure and verifiable manner while complying with the performance and scalability requirements?*

4 Overview

As a solution, we present Wallet, a lightweight confidential computing system for secure serverless deployments in untrusted cloud environments.

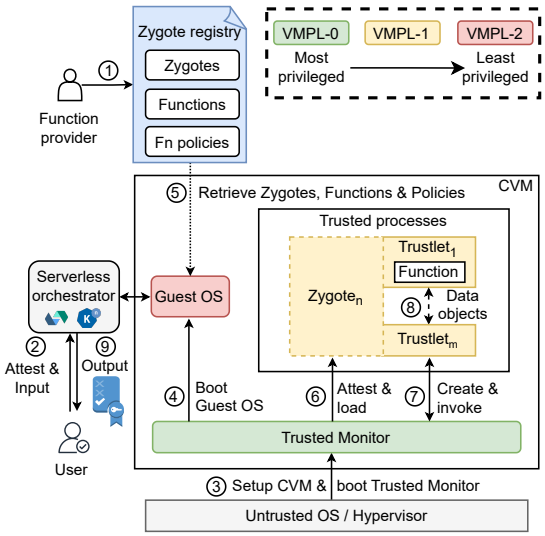

Figure 3: Wallet system architecture overview.

4.1 Threat Model

Wallet extends the CVM threat model [6, 9, 30, 127] by excluding the guest OS from its trust boundaries. We consider adversaries controlling the guest OS attempting to compromise serverless functions within trusted processes, the host OS attacking the CVM via VM-VMM interfaces (e.g., network I/O), or deploying malicious functions to access neighboring functions and leak data. Users trust function providers, but serverless and platform providers are potentially malicious.

The trusted monitor is Wallet's *only* trusted software, enabling remote attestation. We assume that the platform hardware/firmware functions correctly and deployed functions do not intentionally leak data. Users must employ encrypted protocols (e.g., TLS) for communication. Wallet does not address denial-of-service, physical, and side-channel attacks.

4.2 System Architecture

Wallet builds on CVMs and uses hardware-enforced privilege partitioning (e.g., VMPL [10], TD-partitioning [72]) to create lightweight *trustlets* — minimal serverless processes that execute within a secure environment inside a CVM. Figure 3 presents Wallet's key components: the trusted monitor, the trusted processes, the data objects and the zygote registry.

The **trusted monitor** is a small, privileged, and nested software component within Wallet's TCB. It manages the instantiation and invocation of serverless functions within lightweight protection domains isolated from the guest OS. It performs security-critical tasks such as memory management, function deployment, and attestation.

Within the CVM, Wallet introduces the notion of **trusted processes** for hosting serverless functions. These processes possess enclave-like features: *(i)* isolation from the guest OS, *(ii)* trusted boot, and *(iii)* remote attestation capabilities. They are categorized into *zygotes* and *trustlets*. Zygotes serve as pre-initialized templates containing runtime environments (e.g., Python), while trustlets are lightweight instances, derived from zygotes, to execute functions in an isolated manner. To

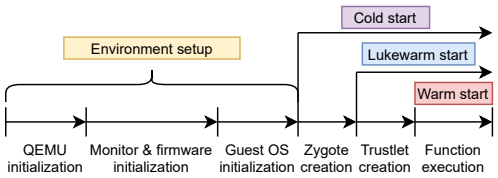

Figure 4: Wallet invocation phases.

facilitate inter-function communication, Wallet provides the abstraction of **data objects**, managed by the trusted monitor ensuring isolation. This design favors reduced boot times, efficient memory use via copy-on-write [5, 52, 53, 96, 116], and optimized data exchange between co-located functions.

Meanwhile, the untrusted guest OS runs at a lower privilege level and mediates communication between the serverless orchestration framework and the trusted monitor. It lies out of the TCB and is responsible for retrieving zygote images and functions from the **zygote registry** and providing them to the monitor for validation, loading, and execution.

4.3 Life of a Request

Figure 3 illustrates the life of a request in Wallet. First, the serverless provider publishes a zygote image with the function runtime in the zygote registry. Function providers develop functions and associate them with a zygote (①), and generate a function key pair for user request encryption.

Initial environment setup. For a function request, users obtain the function provider's public key, encrypt their request where they include a symmetric key for the encryption of the result, and submit it to the serverless orchestrator (e.g., OpenWhisk [48], Knative [43]) (②). This allows the function to directly encrypt with the key of the function provider. The orchestrator checks the available Wallet instance, and if none, launches a CVM with the trusted monitor (③). The trusted monitor then loads the guest OS, where the serverless management system runs, into a lower privilege level (④). Wallet instances can be created proactively by the serverless provider, and its expensive initialization step occurs only once.

Handling function request. Once initialized, the guest OS handles the incoming request, forwarded by the monitor. Wallet has three types of request invocation: *cold*, *lukewarm* and *warm* (Figure 4). The cold start breakoff point in Wallet's workflow is right after the guest OS initialization. The trusted monitor and guest OS constitute the environment for a function execution runtime in the same way a host OS is the environment for the Docker runtime. A cold start occurs when the required zygote is not loaded into the trusted monitor. In this case, the trusted monitor fetches the zygote and function from the registry (⑤). Then, the function provider attests Wallet, establishes secure communication with the trusted monitor, and shares their secrets, including the function's private key and measurements of the zygote image and the function. The guest OS then instructs the trusted monitor to load the zygote (⑥), which is performed after verifying the zygote's integrity against the provided measurement.

At this point, Wallet can process requests. If the function's

Table 2: Monitor system calls *define the interface between the guest OS and the trusted monitor (excerpt).*

Category	API	Description
Zygote	createZygote(void* image) -> zHandle	Loads and attests an image of a zygote and returns a zygote handle.
	deleteZygote(zHandle)	Removes the zygote and its derived trustlets.
Trustlet	createTrustlet(zHandle, fn) -> tHandle	Creates a trustlet based on a zygote and a provided function.
	deleteTrustlet(tHandle)	Deletes the specified trustlet.
	invokeTrustlet(tHandle, void*) -> Result	Run a function in a trustlet and return the result.
Attestation	attestMonitor() -> Report	Returns the attestation report of the monitor.
	attest(Handle) -> Report	Retrieves the attestation report of a trusted process.
Policy	loadPolicy(void*)	Loads the encrypted function provider policies.

zygote exists (*lukewarm start*), WALLET spawns a trustlet for function execution (⑦). Subsequent invocations within a running trustlet (*warm start*) execute without setup delays. The guest OS forwards user data to the trusted monitor, which creates input data objects, decrypts the data, and invokes the trustlet. During execution, the trustlet reads from the input and creates output data objects, which can act as input to other functions, thus enabling efficient function chaining (⑧). Importantly, the trusted monitor mediates this access to ensure isolation between co-located trustlets. Once execution completes, the monitor encrypts the result from output data objects with the request's symmetric key and returns it alongside a signed attestation report to the user via the guest OS (⑨).

4.4 Design Principles and Primitives

We now present the design principles and primitives of WALLET that address the six key problems, outlined in § 3.

#1: Nested confidential execution. As a countermeasure to the large TCB, the hardware-limited scalability and the costly inter-function communication of CVM-based serverless deployments, WALLET introduces nested confidential execution. Inspired by nested virtualization [24, 32, 83, 109, 164]—not directly applicable to CVMs due to host VMM support requirements [4, 66]—WALLET leverages intra-CVM isolation [1, 72, 110] to partition runtime components [104]. This allows the secure coexistence of untrusted components in lower-privilege compartments, while critical services operate in highly privileged domains, reducing the software TCB.

In this model, the *trusted monitor* governs the nested execution environments and operates at the highest privilege level, managing security-critical tasks while restricting untrusted components to lower levels via strict, hardware-enforced page-level access control [1, 72]. Further, it optimizes inter-function communication with data exchange via shared memory, realizing fast chaining for co-located functions.

Nested confidential execution also improves scalability without compromising security. It enables higher function density within a single CVM compared to conventional deployments, which are constrained by hardware limits on concurrent CVM instances per node [15, 73, 75, 125].

#2: Decoupled guest OS architecture. Conventional CVMs include the, often bloated [65], guest OS in their TCB. WALLET decouples the *guest OS* from trusted operations and removes its boot and measurement process from the function invocation path, mirroring ARM TrustZone's principles, i.e., secure/normal world separation [18, 115], as performed in prior works, such as TLR (Trusted Language Runtime) [133].

After trusted monitor initialization, WALLET boots the untrusted guest OS, totally isolated from function execution environments. This decoupling excludes the guest OS kernel from the TCB, significantly reducing the attack surface and ensuring the guest OS vulnerabilities do not compromise the security of serverless functions. The guest OS handles *only* auxiliary tasks such as receiving requests and forwarding function invocations to the trusted monitor, maintaining compatibility with existing serverless frameworks. This reduces startup times compared to CVM-based deployments where cold starts involve full guest OS initialization and measurement.

#3: Trusted process templates. Existing confidential serverless deployments must initialize each function instance from scratch. Startup optimizations (e.g., snapshotting) are impractical [65] due to memory confidentiality (e.g., encryption) requirements. To this end, WALLET introduces *trusted processes* as the core abstraction for executing serverless functions.

These processes—categorized as *zygotes* and *trustlets*, inspired by Android's process model [37, 89]—operate at a dedicated privilege level within isolated address spaces. Zygotes serve as pre-initialized templates with runtime dependencies, while trustlets instantiate from zygotes using copy-on-write mechanisms [5, 52, 53, 96, 116], enabling concurrent function launching with efficient memory usage and high density.

Building on this execution model, WALLET implements its *differential attestation* protocol, which incrementally builds cumulative trust chains by reusing zygote and function measurements, requiring only the measurement of mutable components (e.g., input/output) during invocation, thus reducing attestation latency for both standalone and chained functions.

#4: Dynamically loadable LibOS architecture. To minimize function startup latency and its TCB while supporting diverse workloads (e.g., Python, Node.js), WALLET employs a dynamically loadable LibOS architecture [137, 166], adopting concepts of unikernel-based systems [100, 124, 166]. However, instead of requiring developers to build custom LibOS images, WALLET dynamically loads lightweight LibOS-based runtimes tailored to specific functions at runtime. By providing the language runtimes that directly call a function, common serverless functions can be immediately used inside WALLET.

The LibOS forms the core of WALLET's trusted processes, providing them with essential runtime functionalities. By dynamically loading minimal runtimes rather than full OSes or containers, WALLET reduces initialization overhead.

This design allows on-demand instantiation of lightweight, pre-measured templates and avoids redundant software measurement steps during function invocation, further lowering startup latency without increasing developer effort.

#5: Confidential networking architecture. CVM I/O stacks [44, 117] suffer from performance penalties due to data copies, VM exits, and high CPU utilization for cryptographic operations [62, 92, 112]. Wallet's key insight is to leverage its function density (§ 3.5) and provide a monitor-mediated, high-performance networking architecture for CVMs.

Wallet leverages *function co-location* and employs a *data-centric* I/O approach [161] to optimize function communication via secure data objects, building on insights from prior work (e.g., CAP-VM [135], Nephele [99], Pheromone [161]). Wallet's high function density increases co-location opportunities, and thus, it can also benefit from data-centric schedulers [2, 82, 161] that effectively co-locate related functions.

In Wallet, *data objects* serve as secure endpoints for trustlets to exchange data without traversing conventional I/O stacks. The trusted monitor manages data object allocation and page table permissions, granting appropriate access to producers and consumers, and ensures secure data exchange while maintaining isolation between functions. This design eliminates unnecessary cryptographic operations and redundant data copies, reducing I/O overheads. For functions that cannot be co-located, Wallet falls back to standard CVM I/O, ensuring compatibility with existing serverless frameworks.

5 Design

5.1 Trusted Monitor

The trusted monitor is Wallet's core component. It maintains strict isolation between each serverless function and the guest OS through hardware-enforced CVM partitioning and page table configurations. It operates at the highest privilege level of the CVM partitioning domains (e.g., VMPL-0).

Trusted process management. Wallet's trusted monitor manages the CVM resources and preserves the state of the guest OS and trusted processes using process descriptors. A process descriptor includes the values for general-purpose and control registers, address space information (page tables), process ID, CVM-partitioning privilege level, object information (§ 5.4), and process state. The trusted monitor is responsible for allocating resources and scheduling the guest OS and the trusted processes. Internally, the descriptor state is linked to one vCPU state of the CVM, and the trusted monitor uses a special instruction to switch the context (e.g., AP creation call in SEV-SNP [14]). By executing the guest OS and the trusted processes at lower CVM partitioning levels, the trusted monitor governs their execution.

Isolation enforcement. The isolation between trusted processes and the guest OS requires careful design due to limited isolation domains (e.g., four domains for VMPL [10], three nested guests for TD-Partitioning [72]). To address this issue, the trusted monitor employs conventional ring protection, orthogonal to CVM partitioning: user functions run at ring3, monitor-managed code at ring0, preventing manipulation of the trusted processes' page tables from ring3, and the guest OS at a lower privilege domain (i.e., VMPL-2). The page table-based isolation enables memory optimization through page sharing among functions—impossible in common CVMs—facilitating fast fork-based trustlet creation (§ 5.3.1) and efficient inter-function communication (§ 5.4).

Programming interface. As its execution environment base, Wallet adopts a LibOS architecture [23,27,31,54,91,121]. This aligns well with the requirements of trusted processes, where the main code runs at the ring3, and offers users maximum flexibility through a common POSIX-compatible interface with a minimal resource footprint. Wallet tailors the LibOS architecture for serverless execution by allowing the dynamic loading of functions without requiring the recompilation of other components. Furthermore, the LibOS provides a data-centric I/O [161] API for inter-function communication (§ 5.2) and relies on function provider policies that include workflow information (e.g., function chaining), measurements of the involved zygotes and functions, and, optionally, remote storage encryption keys for functions requiring external storage access.

Monitor system calls. A serverless framework, running within the guest, interacts with the trusted monitor through the defined monitor system calls (Table 2). These calls facilitate the execution of serverless functions, including the creation of zygotes/trustlets, function invocation, performing attestation, and loading the provider's policy. Additionally, the trusted monitor defines an interface between trusted processes for the LibOS to request backend operations (§ 5.2). All interfaces are kept small and concise to eliminate the risk of possible attack vectors introduced by internal oversight.

Attestation service. The trusted monitor includes an attestation service that generates an attestation report for remote users to verify the integrity of Wallet and its function execution. On function execution, the attestation service calculates the measurement of function components following Wallet's differential attestation principles (§ 5.5) to optimize the measurement process by caching previously computed results. The root-of-trust of Wallet relies on CVM's attestation mechanism [14, 74], whose attestation report includes the measurement of the initial guest state and memory. The attestation service integrates this report into its final result.

Performance consideration. The trusted monitor is a central part of Wallet's operations. To ensure performant operations with multiple functions, the monitor allows each I/O operation to run simultaneously for multiple trustlets, without compromising its security properties. For memory accesses, the monitor is only called when (de)allocating memory. It manipulates the trustlets' page table and sets the correct VMPL levels for the new pages, an operation which can also be parallelized.

5.2 LibOS Architecture

Wallet's LibOS [23, 27, 31, 54, 91, 121] integrates a function runtime as well as a shim layer and filesystem with a nested

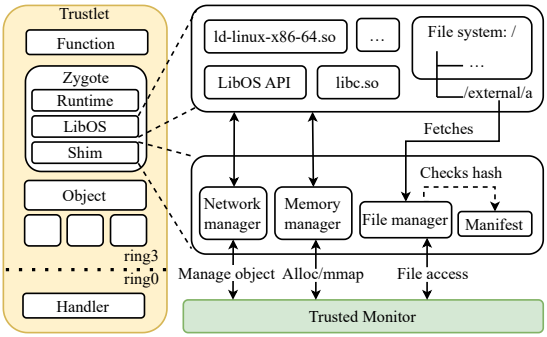

Figure 5: Trusted process architecture.

namespace for efficient function execution.

Shim layer. The shim layer acts as the LibOS backend. While the LibOS provides most required OS services, certain operations mandate assistance from the trusted monitor and guest OS. The shim layer handles three key tasks: *(i)* memory allocation for trusted processes by coordinating with the trusted monitor, *(ii)* reading external host files, and *(iii)* handling data objects for function communication.

Filesystem. The LibOS provides an in-memory file system backed by a zygote, at the cost of a bloated image. Removing non-commonly-used files reduces image size but compromises serverless function flexibility and user experience. As a solution, Wallet employs a *nested namespace* [149] that integrates an embedded with an external filesystem. From the application's perspective, all files appear at the same mount point (e.g., '/'). Internally, the LibOS first checks the embedded filesystem; if a file is missing, it retrieves it from the guest OS's filesystem via the monitor. To ensure file integrity, the shim layer verifies files from the guest OS using provided measurements specified in a manifest [68, 153] embedded in the zygote image and aborts operations in case of a mismatch.

Data-centric I/O. For inter-function communication, the LibOS provides a *data object* abstraction to the trusted processes. This abstraction allows a function to create, get, and update data objects via the API shown in Table 3. Unlike conventional CVM networking that requires traversing entire I/O stacks with multiple context switches and encryption overhead [30, 62, 92, 112], Wallet employs a shared memory-based local object store [29, 86, 120, 122, 131, 161], enabling faster communication. This is especially effective for Wallet as it avoids the costly CVM network overhead (§ 3.3). When the serverless orchestrator fails to co-locate functions, the trusted monitor follows a hybrid I/O approach [101, 161] and resorts to the standard CVM I/O primitives via the guest OS, similar to the external file system access described above. Importantly, this mechanism ensures transparent communication between functions, regardless of whether they are co-located or distributed. § 5.4 details Wallet's communication mechanism.

5.3 Trusted Process

A trusted process (Figure 5) is designed to efficiently host serverless functions within an isolated domain. Each trusted process encapsulates the LibOS, which is independent of the guest OS and provides functionalities to host a serverless runtime. Trusted processes run within the same CVM privilege domain, enabling memory sharing and CoW-based process creation. Meanwhile, the trusted monitor ensures their strict isolation by controlling the per-process page tables. Trusted processes are categorized into two types: *zygotes* and *trustlets*.

Table 3: LibOS API for data objects.

API	Description
createObject(*len,type*) → *obj_id*	Create data object.
getObject(*obj_id*) → **obj*	Get a pointer to the object *obj_id*.
getInputObject() → (*obj_id, len*)	Get data object that contains input.
setOutputObject(*obj_id*)	Set data object as output.

5.3.1 Zygote

A *zygote* is a template process containing a function execution environment tailored for a serverless runtime. Rather than running functions itself, it serves as a base for the instantiation of separate function execution contexts, namely trustlets (§ 5.3.2). They are fetched from the zygote registry, and after the successful integrity verification by the attestation service, the trusted monitor initializes their process descriptor and page tables. To optimize the function invocation process, a zygote offers a pre-initialization mechanism, enabling it to preload and initialize the function runtime environment [26]. During the zygote loading phase, the trusted monitor executes the zygote, and the zygote notifies about the completion of the initialization by making a specific monitor call. The trusted monitor then seals the initialized state and marks all zygote memory non-writable for the subsequent creation of trustlets.

5.3.2 Trustlet

A trustlet serves as the execution environment for serverless functions, inheriting the memory layout and execution environment from its base zygote. Each trustlet has data objects for inter-function communication. On trustlet creation, the trusted monitor duplicates the zygote's process descriptor, loads the function, and inputs data to the corresponding memory region—all without copying the zygote memory.

Copy-on-write mechanism. Any trustlet *write* attempt to the non-writable zygote memory results in a page fault. To handle page faults, the trusted monitor installs a page fault handler for each trustlet. This handler delegates the actual page fault handling to the trusted monitor via a monitor call. Subsequently, the trusted monitor copies the page, updates the trustlet's page table, and then resumes the execution.

Input and output. Function input and output are represented as special types of data objects. A function interacts with these objects via the API in Table 3. When invoking a trustlet, the trusted monitor creates an object for its input and retrieves the results from its output object after its execution.

Scheduling. The guest OS initiates the scheduling of trustlets via the invokeTrustlet() call. Wallet runtime enforces a run-to-completion execution model, i.e., Wallet executes a trustlet until completion for efficient execution [168], and returns the result to the guest OS. This fits for serverless

workloads, which are most of the time short-lived. In case a function performs I/O tasks, the trustlet returns control back to the monitor and a different trustlet can be scheduled, even before the original one has finished its execution. On top of that, the scheduler is also responsible for handling the function chaining of the co-located trustlets (§ 5.4).

5.4 Confidential Network Architecture

Trusted-monitor-mediated I/O path. Data objects constitute a fundamental component of the inter-function communication in Wallet. Figure 6 illustrates their control and data path. In this example, Trustlet1 sends data to Trustlet2 via a data object. First, Trustlet1 invokes the createObject() API call (①). Internally, the trusted monitor allocates the object (②) and updates the Trustlet1's page table to grant it write access (③). Subsequently, when Trustlet2 gets the object via getObject() (④), the trusted monitor updates the Trustlet2's page table, granting it read permission (⑤). Through this trusted monitor-mediated shared-memory communication mechanism, Wallet establishes a secure, yet efficient, channel between functions while maintaining strict isolation. Further, the monitor ensures that each object is attached to only two trustlets at most, one with read and one with write access.

Function chaining. Chaining multiple functions is a common pattern in serverless computing [36, 147, 162]. Wallet's data object enables efficient function chaining through shared memory [122, 131]. Specifically, when chained functions are scheduled on the same host, the trusted monitor creates a special data object and grants write permission to the first function and read permission to the second function. If the functions can not be co-located, Wallet falls back to the normal network path: it returns the result of the first function to the serverless orchestrator, and then the orchestrator invokes the second function.

Note that, these chains are not limited to two functions and could potentially result in a circular chain, leading to the functions not producing a result. The current Wallet prototype does not consider this case. However, circular chains can be detected and prevented by the monitor in a future version. or Wallet's policies could be extended to explicitly disallow it.

External network I/O. To facilitate network communications in Wallet across the CVM boundaries, additional steps are required. The monitor delegates the external network I/O handling to the guest OS. In practice, the monitor should encrypt the network data and provide it to the serverless framework for forwarding. Then, the serverless framework can send this data to the desired destination and get a response.

5.5 Differential Attestation

Wallet introduces a *differential attestation protocol* to reduce the latency of verifying serverless function execution, which requires attesting the platform, runtime, function, and I/O. This protocol retains prior measurements to compose a cumulative report and measures only the mutable parts at runtime.

Security guarantees. Wallet's differential attestation ensures the integrity of the monitor, trustlet, zygote, function, its input, and output. It relies on the correctness of Wallet, in particular on Wallet's monitor, which is the only permanently immutable component of the system. All other parts of the differential attestation are at one point mutable, since they are fetched and instantiated by the monitor. After their initialization, all parts that constitute a trustlet become immutable with data protected with CoW. This enables the monitor to cache the hashes of these components for future use.

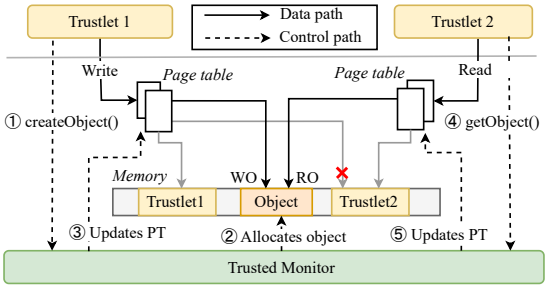

Figure 6: Inter-function communication with data objects.

Attestation flow. During **initialization**, the function provider shares the public function key with the user and establishes a secure connection with Wallet, which relies on the attestation of the trusted monitor (rooted in ASP [14]). The provider sends a nonce to the monitor to prevent replay attacks, which responds with its attestation report. After verifying the monitor, the provider sends the function's private key and a policy to Wallet, which are used for subsequent decryption and attestation of trusted processes, respectively.

Importantly, before providing private or security-sensitive data as a function input, a user must first perform a test function call, retrieve the attestation report, and verify that the Wallet instance is at the expected state and the correct toolchain is present. After this point, the user has gained trust in Wallet and can proceed with their function invocations.

The **function invocation** phase handles *cold*, and *warm* starts (Figure 4). In a cold start, the attestation service verifies the integrity of trusted processes' components against the measurements in the provider's policy. Subsequent invocations of the same function (warm starts) only require measuring the function I/O, building upon prior measurements of the monitor, zygote, and function. For chained functions, the extended report includes all their measurements, reducing attestation time while ensuring end-to-end verifiability. We describe the differential attestation protocol in detail in § B.1.

6 Security Analysis

We present a summary of our security analysis and formal verification of Wallet's security protocol, with further details provided in § A and the complete proof detailed in § B.2.

Attack vector analysis. Table 4 summarizes potential attack vectors and Wallet's mitigations. The guest OS cannot access trusted monitor/process memory or modify VMPL configurations due to its lower privilege level (VMPL-2). Further, data in serverless requests remains encrypted

Table 4: Potential attack vectors and WALLET's mitigations.

Attack vector	Mitigation
From guest OS	
Read the WALLET's memory	VMPL protection
Read user data in the request	Data encryption
Load crafted zygotes/functions/inputs	Attestation
Load compromised external file	Measurement check
From host and hypervisor	
Access VM's memory	CVM protection
DMA to the VM's memory	CVM protection
Inject malicious interrupts [138, 139]	Alternate injection [9]
Return invalid CPUID values [93]	CPUID pages [14]
Launch compromised images	Attestation
Manipulate user request	Protocol encryption
From co-located functions	
Access other trustlet's memory	Page table protection
Reuse-based attacks [167]	Recreate trustlets

with user-specific keys exclusively accessible to the trusted monitor, preventing unauthorized access. WALLET's attestation service prevents arbitrary zygote/function loading by validating measurements against function providers' policies, while users can detect replay attacks by verifying attestation reports. File tampering in the nested filesystem is thwarted by LibOS-enforced measurement checks.

Against host/hypervisor attacks, AMD SEV-SNP's reverse map page (RMP) table blocks access to CVM private memory, while hardware prevents DMA attacks. Further, SEV-SNP's alternate interrupt injection [9] counters malicious interrupt/exception injection attacks [138, 139], while its secure CPUID page prevents invalid CPUID value attacks [93]. Tampered trusted monitor deployments or attempts to bypass it are detected via invalid measurements in attestation reports, ASP-signed certificates prevent forged reports, and TLS must be employed to secure the network communication.

Co-located function attacks are prevented by trustlets operating in ring3 with dedicated trusted monitor-managed page tables that prohibit cross-trustlet memory access. Reuse-based attacks [167] are mitigated by recreating trustlets for different users' requests, eliminating potentially compromised, residual state from prior executions.

Formal verification of the security protocols. We formally verify WALLET's differential attestation protocol using the Tamarin Prover [50, 105] under the Dolev-Yao [51] attacker model. In our model, we treat cryptographic functions (e.g., hashing) as perfect and messages as atomic. We also assume the correct functionality of the report verification infrastructure and the underlying hardware, i.e., the ASP does not leak its secrets and produces correct reports.

The verification considers two key properties for WALLET's differential attestation protocol: *(i)* **secrecy**, secrets (e.g., encrypted data, private keys) remain undisclosed unless explicitly compromised; and *(ii)* **authenticity**, guaranteeing that attested results correspond to correct function execution with validated inputs. Tamarin confirmed that no execution trace violates our properties, proving secrecy and authenticity.

7 Evaluation

We evaluate WALLET by analyzing its end-to-end performance (§ 7.2),the performance of its operations (§ 7.3), its resource efficiency (§ 7.4), its communication network performance (§ 7.5), and its scale-out capabilities (§ 7.6).

7.1 Experimental Setup

Testbed. We perform our experiments on an AMD SEV-SNP-enabled server with an AMD EPYC 7713P CPU (64 cores, hyperthreading disabled) and 1024 GB of DDR4 DRAM (16×64 GB/DIMM). The server runs NixOS 24.11 with an AMD SEV-SNP-enabled Linux kernel (v6.8.0). Each VM uses an Ubuntu-22.04 with a VMPL-enabled Linux kernel (v6.5.0).

Variants. We use the baselines summarized below.

Variant	Execution environment
Native	Bare-metal instance on Linux
LibOS (Gramine)	Gramine LibOS on Linux
Container (Kata)	Kata containers runtime with QEMU/KVM
VM (KVM-Linux)	Standard VM with Linux guest OS
CVM (SEV-SNP)	AMD SEV-SNP VM with Linux guest OS
WALLET	Our WALLET system

Workloads. For performance evaluation, we use Python functions from the SeBS benchmark suite [34]. The current WALLET prototype lacks external network and *fork()* support; therefore, we exclude "uploader", "crud-api" (requires external networking), and "video-processing" (requires running *ffmpeg* as a standalone process). Additionally, we conduct a simulation-based study with Azure Functions production traces [142, 165].

7.2 End-to-end Performance

Methodology. We run the SeBS server in each variant and co-locate the client on the same host. WALLET does not include the time of measurement calculation for a fair comparison with the other baselines. We run each experiment *five* times and report the average.

End-to-end latency. Figure 7 presents the end-to-end client latency of each SeBS function for each variant. In the cold start case, WALLET (Lukewarm) achieves 93.44% speedup over CVM (SEV-SNP) on average, and is 83.99%, 51.04%, 32.85% faster than VM (KVM-Linux), Containers (Kata), and LibOS (Gramine). Even in the "cold"est case (i.e., no loaded zygote), WALLET is on average 85.10% faster than the CVM (SEV-SNP) baseline and 63.62% faster than the VM. When compared to the Containers (Kata), and LibOS (Gramine), WALLET incurs 11.25% and 52.57% performance overhead, respectively. In the warm start case, the performance difference among variants becomes smaller as the existing function environment is reused. Still, WALLET is 14.97%, 7.53% and 1.04% faster than the CVM, VM and Containers (Kata) case respectively while it shows a 15.63% performance overhead relative the LibOS (Gramine). WALLET's performance gains mainly stem from its efficient function startup, run-to-completion execution model that minimizes VMEXITs, and in-memory filesystem. WALLET exhibits higher latency if the input/output size is large due to data transfer overhead between different VMPL levels (e.g., image-recognition has 100MB input) (§ 7.3).

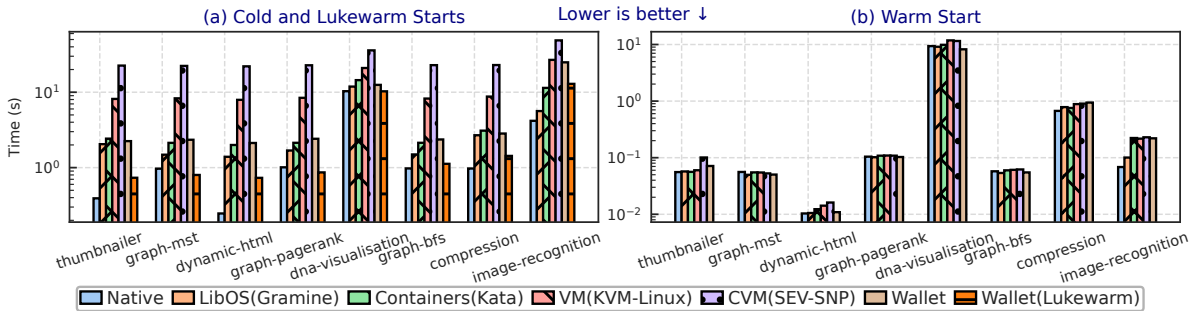

Figure 7: SeBS benchmark end-to-end latency: (a) cold and lukewarm starts, and (b) warm start.

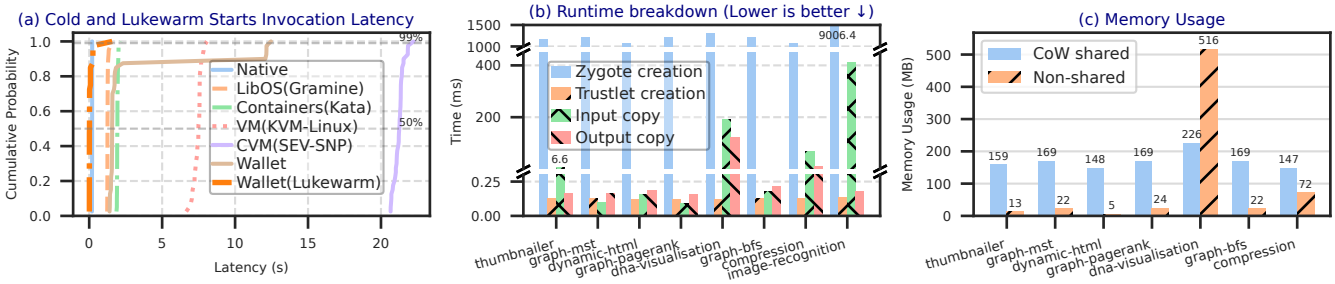

Figure 8: Performance analysis: (a) invocation latencies, (b) Wallet's runtime breakdown, and (c) Wallet's memory usage.

Invocation latency. Figure 8a shows the cumulative distribution function (CDF) of invocation latency under cold starts, which primarily affect the invocation latency. Wallet (Lukewarm) has p50/p99 (50th and 99th percentile) latencies of 0.019/1.06 s. It achieves ms-scale p50 startup latency, comparable to the latest (non-confidential) fork-based serverless systems [26, 52, 53, 96, 116]. On the other hand, Wallet (cold) has 1.57/12.34 s latency, which is at the levels of Containers (1.93/2.10 s) and much lower than the CVM (21.25/22.10 s) and the VM (7.5/8.0 s) variants. It also exhibits a larger standard deviation (3.51 s), which originates from the time needed to load the zygote and the variably-sized functions (§ 7.3).

7.3 Performance Analysis

Boot time analysis. Figure 1a presents the boot time breakdown. For VM-based variants, the majority of the boot time is consumed by VMM (QEMU) initialization and guest OS startup. The CVM incurs additional overhead due to extra management tasks (e.g., measurement of initial state, memory validation) [112]. Although Wallet's environment setup takes 4.05 s longer than the CVM (without memory preallocation (§ 7.3.1)) because of the added setup time for the monitor, Wallet's cold start only takes 2.38 s by eliminating the CVM initialization time. Wallet's lukewarm start further optimizes the boot time by employing a fork-based start (10.3 ms).

Runtime processing cost. Further, we analyze the runtime processing of Wallet, focusing on zygote and trustlet creation, and input/output transfer. Each process involves transferring data between the guest OS and the trusted monitor, followed by updating the VMPL level and the trustlet's page tables. As Figure 8b illustrates, the zygote creation is responsible for most of the runtime initialization. The current prototype bundles required libraries in the base image (e.g., numpy, pytorch), increasing the zygote size, ranging from 60 MB to 691 MB (image-recognition) in SeBS. However, zygote creation occurs only during cold starts, and the CoW mechanism enables efficient memory sharing among trustlets using the same zygote image (§ 7.4).

On the other hand, the trustlet creation and data transfers' duration depends on the function and I/O size. Each SeBS function is less than 4 KB. Thus, the trustlet creation requires < 0.2 ms. However, larger data transfers, such as in dna-visualisation (112 MB output), take ~ 122 ms. If the data fits within a 4K page, the copying time is <0.1 ms (e.g., graph-mst and graph-pagerank).

Measurement cost. Figure 2c shows the breakdown of the time taken to calculate the SHA512 measurement of an empty Python function. The calculation time is proportional to the data size. In the cold start case, the measurements include those of the trusted monitor, as well as the zygote (60 MB), which takes around 1.1 s. The current prototype does not support CPU acceleration for SHA, justifying the lower performance. However, for the lukewarm start, Wallet only needs to measure the function, input, and output (typically less than 4KB), dropping the calculation time below 0.5 ms.

7.3.1 Effectiveness of Optimizations

Memory preallocation. Memory validation [14] is expensive as it involves VMEXIT for state updates. In our setup, validating a 4K page takes 24 µs (6ms/MB). Although this cost is an one-time overhead, it affects the initial creation time of zygotes and trustlets. Memory preallocation removes this validation cost at runtime in exchange for an increased bootup time. For the SeBS benchmark, the preallocation improves the performance by 49.22% in the cold start and 4.72% in the warm start case on average, over the non-preallocation

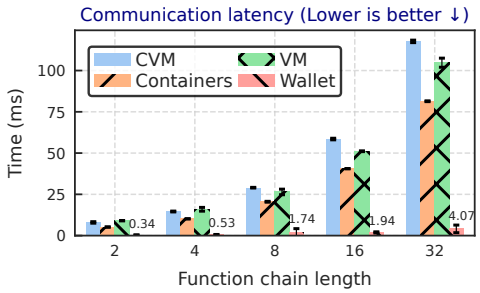

Figure 9: Latencies of function chains of different lengths.

version. However, preallocating 16 GB of memory increases the boot time by 238 s, highlighting the trade-off.

Copy-on-Write (CoW). We further investigate the impact of CoW. Performance-wise, CoW is crucial for achieving rapid trustlet creation and, thus, fast lukewarm starts (Figure 8b). CoW reduces the average trustlet creation time of SeBS functions from 66 ms to 0.11 ms. However, it incurs additional overhead during execution due to page faults. This overhead becomes noticeable mostly in the warm starts, but still remains almost negligible (~1%), promoting CoW's adoption.

7.4 Resource Efficiency

Memory footprint. We evaluate the memory footprint of Wallet while running SeBS functions. Figure 8c presents the result. "CoW shared" represents the memory shared with CoW among trustlets and their base zygote, whereas "Non-shared" denotes the memory exclusively allocated for a trustlet during execution. Although the amount of allocated memory ("Non-shared") varies depending on the workload, on average, trustlets share 170 MB of memory with the base zygote, reducing the total system memory consumption.

Function density. Memory sharing enables Wallet to pack more functions per node. We analyze the memory consumption for increasing numbers of concurrently-running empty Python functions, shown in Figure 2b. We enable kernel same-page merging (KSM) [45] on the host to de-duplicate identical memory pages. For Wallet, the total memory is calculated as the base zygote shared memory (147 MB) plus 60 KB of non-CoW-shared memory per function. As KSM cannot deduplicate encrypted memory, CVM memory consumption is proportional to the number of functions, leading to a higher per-function memory overhead. Further, the number of CVMs in a single node is limited to the number of encryption keys (509 in our environment). In contrast, Wallet achieves significantly lower memory usage, realizing high function density.

7.5 Communication Analysis

Methodology. To evaluate Wallet's inter-function communication, we measure the communication time of invoking multiple functions in a chain by varying the data size and the number of functions in the chain. All functions are deployed on the same host. In Wallet, each function creates its communication endpoint for input and output, while other baselines use TCP networking.

Communication latency. First, we show the communication latency between two functions under various data sizes. Figure 1b shows the result. This result indicates that Wallet achieves lower latency than CVMs (1.4-27×), as well as traditional VMs and Containers (Kata), across message sizes, thanks to the zero-copy-based data objects.

Latencies of function chains. Next, we show the latencies of function chains. As shown in Figure 9, Wallet outperforms all baselines across all chain lengths, achieving 16.7-30.2×, 11.8-20.9×, and 15.2-30.1× lower communication latency than CVMs, Containers, and traditional VMs, respectively. Wallet's performance advantage grows with chain length, demonstrating the efficiency of its data-centric I/O for inter-function communication compared to TCP-based networking used by other approaches, highlighting Wallet's suitability for complex workloads with function chaining.

7.6 Scale-out Performance

Methodology. We develop a simulator modeling real-world serverless deployments with a central scheduler and multiple nodes, each having fixed execution slots and LRU caches for warm functions. The scheduler processes trace data chronologically, prioritizing nodes already caching the requested function. If none are available, it assigns requests to free nodes or queues them. Function execution duration gets increased based on its boot type (cold/lukewarm/warm), using timings sampled from our microbenchmarks (Figure 1a). The lukewarm boot, unique to Wallet, occurs when a node caches another function from the same application, simulating zygote sharing. We use Azure Functions production traces [142, 165] sampled with InVitro [154], comprising 4k functions with ~4.1 million invocations over 30 minutes.

Invocation latency and per-function slowdown. We run the simulator with 100 nodes, each having 32 execution slots and a cache size of 32. Figure 10a shows the CDF of invocation latency: at p50/p99, CVM and VM experience delays of 489/881 s and 1.3/33 s, respectively, while Wallet achieves just 5 ms/1.5 s. Figure 10b reports the per-function slowdown, which increases as function execution time decreases and invocation latency dominates. At p50/p99, CVM slowdowns reach 2,682/391,257, VM slowdowns are 13.98/8,658, while, in contrast, Wallet's slowdowns are merely 1.02/7.06.

Node scalability. We also evaluate the invocation latency for various numbers of nodes. Figure 10c shows that, compared to the other baselines, Wallet achieves lower latency with a smaller number of nodes. For example, Wallet has 5 ms p99 latency with 100 nodes, while, even with 150 nodes, the CVM's p99 latency is 50 s, and the VM's is 12 s. These results highlight Wallet's effectiveness in a large-scale environment.

8 Related Work

Serverless computing. Serverless computing has transformed cloud deployment. Open-source frameworks (OpenWhisk [48], OpenLambda [63], OpenFaaS [47]) and major cloud providers (AWS Lambda [141], Azure Functions [108],

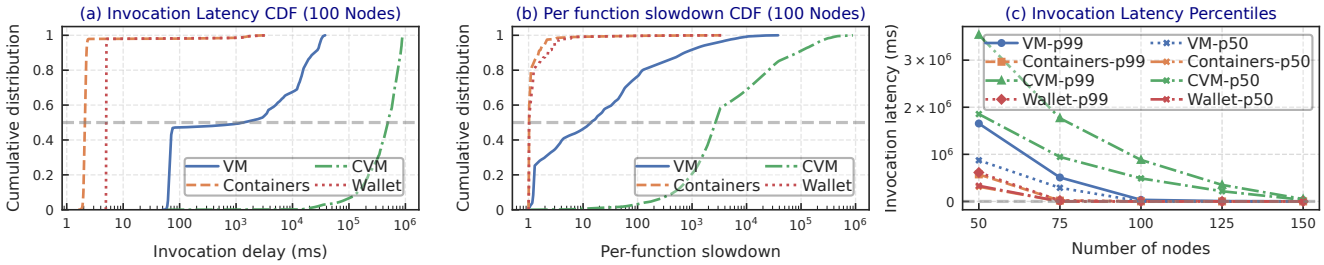

Figure 10: Scale-out results using Azure Functions production traces [142, 165].

Google Cloud Functions [60]) offer widely used architectures. Recent research [5, 26, 53, 80, 81, 116, 128, 132, 144, 146, 165] mainly focuses on improving serverless isolation, performance and efficiency, while Wallet targets the security challenges and leverages TEEs to form a lightweight confidential computing system for secure serverless deployments.

Confidential computing. Confidential computing, based on hardware TEEs [9, 16–18, 76, 77, 90], is widely adopted for protecting data and code in untrusted clouds. TEE technologies can be broadly categorized into process-based (e.g., Intel SGX [76]) and VM-based TEEs (e.g., AMD SEV-SNP [9], Intel TDX [77], ARM CCA [17]). Extensive recent research [4, 11, 21, 56–58, 61, 65, 88, 92, 97, 113, 118, 129, 134, 145, 155, 157, 159, 160, 163, 167] aims to minimize TCB, and provide intra-VM isolation, secure data at rest, among others.

Specifically, Veil [4], and NestedSGX [157] deploy secure services in CVMs using SEV-SNP VMPL. Erebor [163] adopts intra-kernel privilege isolation to provide sandboxing in CVMs. Unlike these, Wallet targets serverless deployments, and leverages CVM partitioning for minimal TCB, fast boot, and efficient function communication via shared memory.

Confidential serverless computing. Many research works aim for secure serverless computing architectures using confidential computing [7, 25, 55, 84, 94, 119, 123, 140, 152]. ServerlessCoCo [140] comprehensively analyzes overheads in CVM-based serverless deployments. Plugin Enclaves [94] proposes a hardware-based approach with Intel SGX for efficient confidential serverless computing. Cryonics [84] reduces startup times using snapshot-based SGX enclaves. In contrast, Wallet materializes confidential serverless computing by providing a CVM-based lightweight system with easily attestable and deployable trusted processes.

Concurrently with our work, COFUNC [143] proposes a library OS-based confidential containers tailored for serverless functions. COFUNC employs a microkernel architecture, running multiple containers with the same CVM while enforcing isolation. Contrary to Wallet, COFUNC offloads container management tasks to the serverless runtime that runs on the host. On the other hand, Wallet allows for the reuse of the existing OS and serverless framework within the CVM, leveraging its nested confidential execution architecture.

Lightweight virtualization. Lightweight virtualization techniques [3, 27, 39, 42, 85, 87, 88, 100, 102] aim to reduce the VM overheads while maintaining isolation. Firecracker [3] introduces microVMs for serverless workloads. Kata Containers [42] deploy lightweight VMs for container applications, whereas Unikraft [87] creates specialized unikernels for applications. Gramine LibOS [27, 39, 88] runs unmodified apps in SGX enclaves and TDX VMs [88]. On the other hand, Wallet leverages these concepts with a minimal Gramine LibOS instance for each of its trusted processes, enabling low boot and attestation times, and seamless application deployment. Unlike Gramine-TDX, Wallet's LibOS is only running in a certain privilege level within the CVM and does not have system-wide privileges. Further, in contrast to Gramine's single-tenant model with its existing backends, Wallet provides a framework for managing multiple instances within the same CVM, enabling multitenancy and optimized inter-function communication via data objects among trusted processes while maintaining function instance protection.

Networking/IO for confidential computing. TEE networking is typically slow due to the added overhead of encryption and data copying with multiple context switches [92, 112, 158] Several works optimize their processing overhead with polling at the cost of increased CPU usage [20, 21, 150, 151]. Compared to them, Wallet achieves efficient inter-function communication through its data-centric shared-memory-based I/O architecture while ensuring isolation.

9 Conclusion

In this paper, we present Wallet, a lightweight confidential serverless computing system. Wallet leverages intra-CVM partitioning mechanisms and efficiently consolidates security-critical functionalities into a compact, privileged trusted monitor, resulting in a small TCB. Through its zygote mechanism, Wallet achieves fast boot times while optimizing memory management and reducing the communication cost between serverless functions via its data-centric I/O architecture. Further, it provides a formally verified differential attestation mechanism to ensure end-to-end trust with reduced attestation times. Overall, Wallet introduces low-performance overheads while exposing a minimal attack surface, making it ideal for security- and latency-sensitive serverless deployments.

Artifact and appendix. Wallet is publicly available at https://github.com/TUM-DSE/Wallet-VMPL. The appendix covers the formal verification of security protocols in Tamarin [49], and additional implementation details.

Acknowledgments

We thank our shepherd, Prof. Anurag Khandelwal, and the anonymous reviewers for their helpful comments. We also thank Prof. Nuno Santos for valuable discussion and feedback on this work.

This work was supported in part by an ERC Starting Grant (ID: 101077577) and the Chips Joint Undertaking (JU), European Union (EU) HORIZON-JU-IA, under grant agreement No. 101140087 (SMARTY), the Intel Trustworthy Data Center of the Future (TDCoF), and Google Research Grants. The authors acknowledge the financial support by the Federal Ministry of Research, Technology and Space of Germany in the programme of "Souverän. Digital. Vernetzt.". Joint project 6G-life, project identification number: 16KISK002.

References

[1] Confidential Computing 101. Virtual Machine Privilege Levels, 2024. Last accessed on 2025-03-27. URL: https://docs.enclaive.cloud/confidential-cloud/technology-in-depth/amd-sev/technology/fundamentals/features/virtual-machine-privilege-levels.

[2] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. Palette Load Balancing: Locality Hints for Serverless Functions. In *Proceedings of the 18th European Conference on Computer Systems*. Association for Computing Machinery, 2023. doi:10.1145/3552326.3567496.

[3] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2020. URL: https://www.usenix.org/conference/nsdi20/presentation/agache.

[4] Adil Ahmad, Botong Ou, Xiaokuan Zhang, and Pedro Fonseca. VEIL: A Protected Services Framework for Confidential Virtual Machines. In *Proceedings of the 29th International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2024. doi:10.1145/3623278.3624763.

[5] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the 2018 USENIX Annual Technical Conference*. USENIX Association, 2018. URL: https://www.usenix.org/conference/atc18/presentation/akkus.

[6] Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, and Felix Wilhelm. Intel Trust Domain Extensions (TDX) Security Review April 2023, 2023. Last accessed on 2025-03-27. URL: https://services.google.com/fh/files/misc/intel_tdx_-_full_report_041423.pdf.

[7] Fritz Alder, N. Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. S-FaaS: Trustworthy and Accountable Function-as-a-Service using Intel SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*. Association for Computing Machinery, 2019. doi:10.1145/3338466.3358916.

[8] Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. Secure Serverless Computing using Dynamic Information Flow Control. *Proc. ACM Program. Lang.*, 2(OOPSLA), 2018. doi:10.1145/3276488.

[9] AMD. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More, 2020. Last accessed on 2025-03-27. URL: 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.

[10] AMD. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More, 2020. Last accessed on 2025-03-27. URL: https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/solution-briefs/amd-secure-encrypted-virtualization-solution-brief.pdf.

[11] AMD. Linux SVSM (Secure VM Service Module), 2023. Last accessed on 2025-03-27. URL: https://github.com/AMDESE/linux-svsm.

[12] AMD. Secure VM Service Module for SEV-SNP Guests Guest Communication Interface Revision: 1.00, 2023. Last accessed on 2025-03-27. URL: https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/58019.pdf.

[13] AMD. AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions Revision 3.36, 2024. Last accessed on 2025-03-27. URL: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24594.pdf.

[14] AMD. SEV Secure Nested Paging Firmware ABI Specification Revision: 1.57, 2025. Last accessed on 2025-03-27. URL: https://www.amd.com/content/

dam/amd/en/documents/epyc-technical-docs/specifications/56860.pdf.

[15] AMD. Using SEV with AMD EPYC™ Processors, 2025. Last accessed on 2025-03-27. URL: https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/tuning-guides/58207-using-sev-with-amd-epyc-processors.pdf.

[16] AMD. AMD Secure Encrypted Virtualization (SEV), [n. d.]. Last accessed on 2025-03-27. URL: https://www.amd.com/en/developer/sev.html.

[17] ARM. Arm Confidential Compute Architecture, [n. d.]. Last accessed on 2025-03-27. URL: https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture.

[18] ARM. Arm TrustZone for Cortex-M, [n. d.]. Last accessed on 2025-03-27. URL: https://www.arm.com/technologies/trustzone-for-cortex-m.

[19] Akram Aslani and Mostafa Ghobaei-Arani. Machine learning inference serving models in serverless computing: a survey. *Computing*, 107(1), January 2025. doi:10.1007/s00607-024-01377-9.

[20] Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Do Le Quoc, Vijay Nagarajan, and Pramod Bhatotia. Avocado: A Secure In-Memory Distributed Storage System. In *Proceedings of the 2021 USENIX Annual Technical Conference*. USENIX Association, 2021. URL: https://www.usenix.org/conference/atc21/presentation/bailleu.

[21] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*. USENIX Association, 2019. URL: https://www.usenix.org/conference/fast19/presentation/bailleu.

[22] Amine Barrak, Fabio Petrillo, and Fehmi Jaafar. Serverless on machine learning: A systematic mapping study. *IEEE Access*, 10:99337–99352, 2022. doi:10.1109/ACCESS.2022.3206366.

[23] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, and Galen C. Hunt. Composing OS extensions safely and efficiently with Bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems*. Association for Computing Machinery, 2013. doi:10.1145/2465351.2465375.

[24] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2010. URL: https://www.usenix.org/conference/osdi10/turtles-project-design-and-implementation-nested-virtualization.

[25] Stefan Brenner and Rüdiger Kapitza. Trust More, Serverless. In *Proceedings of the 12th ACM International Conference on Systems and Storage*. Association for Computing Machinery, 2019. doi:10.1145/3319647.3325825.

[26] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proceedings of the 15th European Conference on Computer Systems*. Association for Computing Machinery, 2020. doi:10.1145/3342195.3392698.

[27] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference*. USENIX Association, 2017. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai.

[28] Jiahao Chen, Zeyu Mi, Yubin Xia, Haibing Guan, and Haibo Chen. CPC: Flexible, Secure, and Efficient CVM Maintenance with Confidential Procedure Calls. In *Proceedings of the 2024 USENIX Annual Technical Conference*. USENIX Association, 2024. URL: https://www.usenix.org/conference/atc24/presentation/chen-jiahao.

[29] Qiong Chen, Jianmin Qian, Yulin Che, Ziqi Lin, Jianfeng Wang, Jie Zhou, Licheng Song, Yi Liang, Jie Wu, Wei Zheng, Wei Liu, Linfeng Li, Fangming Liu, and Kun Tan. YuanRong: A Production General-purpose Serverless System for Distributed Applications in the Cloud. In *Proceedings of the ACM SIGCOMM 2024 Conference*. Association for Computing Machinery, 2024. doi:10.1145/3651890.3672216.

[30] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel TDX Demystified: A Top-Down Approach. *ACM Comput. Surv.*, 56(9), 2024. doi:10.1145/3652597.

[31] David R. Cheriton and Kenneth J. Duda. A Caching Model of Operating System Kernel Func-

tionality. *SIGOPS Oper. Syst. Rev.*, 29(1), 1995. doi:10.1145/202453.202476.

[32] Google Cloud. Enable nested virtualization, [n. d.]. Last accessed on 2025-04-04. URL: https://cloud.google.com/compute/docs/instances/nested-virtualization/enabling.

[33] The Confidential Computing Consortium. Confidential Computing: Hardware-Based Trusted Execution for Applications and Data: November 2022, V1.3, 2022. Last accessed on 2025-03-27. URL: https://confidentialcomputing.io/wp-content/uploads/sites/10/2023/03/CCC_outreach_whitepaper_updated_November_2022.pdf.

[34] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Proceedings of the 22nd International Middleware Conference*. Association for Computing Machinery, 2021. doi:10.1145/3464298.3476133.

[35] Lazar Cvetković, François Costa, Mihajlo Djokic, Michal Friedman, and Ana Klimovic. Dirigent: Lightweight serverless orchestration. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. Association for Computing Machinery, 2024. doi:10.1145/3694715.3695966.

[36] Nilanjan Daw, Umesh Bellur, and Purushottam Kulkarni. Xanadu: Mitigating Cascading Cold Starts in Serverless Function Chain Deployments. In *Proceedings of the 21st International Middleware Conference*. Association for Computing Machinery, 2020. doi:10.1145/3423211.3425690.

[37] Android Developers. About the Zygote Processes – Android Open Source Project, [n. d.]. Last accessed on 2025-03-27. URL: https://source.android.com/docs/core/runtime/zygote.

[38] Coconut-SVSM Developers. COCONUT SVSM, [n. d.]. Last accessed on 2025-03-27. URL: https://github.com/coconut-svsm/svsm.

[39] Gramine Developers. Gramine: A library OS for Linux multi-process applications, with Intel SGX support, 2024. URL: https://github.com/gramineproject/gramine.

[40] Gramine Developers. Gramine PAL host ABI, 2024. Last accessed on 2025-03-27. URL: https://gramine.readthedocs.io/en/stable/pal/host-abi.html.

[41] Hyperlight Developers. Hyperlight: a Lightweight Virtual Machine Manager (VMM) Designed to be Embedded within Applications, 2025. Last accessed on 2025-03-27. URL: https://github.com/hyperlight-dev/hyperlight.

[42] Katacontainers Developers. Kata Containers - Open Source Container Runtime Software , [n. d.]. Last accessed on 2025-03-27. URL: https://katacontainers.io/.

[43] Knative Developers. Knative, [n. d.]. Last accessed on 2025-03-27. URL: https://knative.dev/docs/.

[44] Linux Developers. vsock - Linux VSOCK Address Family, 2024. Last accessed on 2025-03-27. URL: https://man7.org/linux/man-pages/man7/vsock.7.html.

[45] Linux Developers. Kernel Samepage Merging – The Linux Kernel Documentation, [n. d.]. Last accessed on 2025-03-27. URL: https://docs.kernel.org/admin-guide/mm/ksm.html.

[46] Linux Kernel Developers. Guest halt polling – The Linux Kernel Documentation, [n. d.]. Last accessed on 2025-03-27. URL: https://docs.kernel.org/virt/guest-halt-polling.html.

[47] OpenFaas Developers. OpenFaaS - Serverless Functions, Made Simple, [n. d.]. Last accessed on 2025-03-27. URL: https://www.openfaas.com/.

[48] OpenWhisk Developers. OpenWhisk - Open Source Serverless Cloud Platform, [n. d.]. Last accessed on 2025-03-27. URL: https://openwhisk.apache.org/.

[49] Tamarin Developers. Tamarin Manual. Last accessed on 2025-03-27. URL: https://tamarin-prover.com/manual/master/tex/tamarin-manual.pdf.

[50] Tamarin Developers. Tamarin Prover, [n. d.]. Last accessed on 2025-03-27. URL: https://tamarin-prover.com/.

[51] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983. doi:10.1109/TIT.1983.1056650.

[52] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Serverless Computing on Heterogeneous Computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2022. doi:10.1145/3503222.3507732.

[53] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen.

Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2020. doi:10.1145/3373376.3378512.

[54] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: an Operating System Architecture for Application-level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, 1995. doi:10.1145/224056.224076.

[55] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the PENGLAI enclave. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2021. URL: https://www.usenix.org/conference/osdi21/presentation/feng.

[56] Xinyang Ge, Hsuan-Chi Kuo, and Weidong Cui. Hecate: Lifting and Shifting On-Premises Workloads to an Untrusted Cloud. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2022. doi:10.1145/3548606.3560592.

[57] Dimitra Giantsidi, Julian Pritzi, Felix Gust, Antonios Katsarakis, Atsushi Koshiba, and Pramod Bhatotia. TNIC: A Trusted NIC Architecture: A Hardware-network Substrate for Building High-Performance Trustworthy Distributed Systems. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. Association for Computing Machinery, 2025. doi:10.1145/3676641.3716277.

[58] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. Acctee: A webassembly-based two-way sandbox for trusted resource accounting. In *Proceedings of the 20th International Middleware Conference*. Association for Computing Machinery, 2019. doi:10.1145/3361525.3361541.

[59] Google. Google Cloud Confidential Computing, 2019. Last accessed on 2025-03-27. URL: https://cloud.google.com/confidential-computing/.

[60] Google Cloud Functions, [n. d.]. Last accessed on 2025-03-27. URL: https://cloud.google.com/functions.

[61] Jinyu Gu, Bojun Zhu, Mingyu Li, Wentai Li, Yubin Xia, and Haibo Chen. A Hardware-Software Co-design for Efficient Intra-Enclave Isolation. In *Proceedings of the 31st USENIX Security Symposium*. USENIX Association, 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/gu-jinyu.

[62] Nadav Har'El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger, and Razya Ladelsky. Efficient and Scalable Paravirtual I/O System. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*. USENIX Association, 2013.

[63] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless Computation with OpenLambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*. USENIX Association, 2016. URL: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson.

[64] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, 2011.

[65] Benjamin Holmes, Jason Waterman, and Dan Williams. Severifast: Minimizing the root of trust for fast startup of sev microvms. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2024. doi:10.1145/3620665.3640424.

[66] Zhichao Hua, Jinyu Gu, Yubin Xia, Haibo Chen, Binyu Zang, and Haibing Guan. vtz: Virtualizing arm trustzone. In *Proceedings of the 26th USENIX Conference on Security Symposium*. USENIX Association, 2017.

[67] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. A Case against (Most) Context Switches. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. Association for Computing Machinery, 2021. doi:10.1145/3458336.3465274.

[68] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.*, 41(2), 2007. doi:10.1145/1243418.1243424.

[69] IBM. Confidential computing on IBM Cloud, 2019. Last accessed on 2025-03-27. URL: https://www.ibm.com/cloud/confidential-computing.

[70] Global Market Insights Inc. Serverless Architecture Market Size - By Service Type (Function as a Service (FaaS), Backend as a Service (BaaS)), By Deployment

Mode (Public Cloud, Private Cloud, Hybrid Cloud), By Organization Size, By Component, By Industry Vertical & Forecast, 2024 - 2032, 2024. Last accessed on 2025-03-27. URL: https://www.gminsights.com/industry-analysis/serverless-architecture-market.

[71] Intel. intel/vtpm-td, 2023. Last accessed on 2025-03-27. URL: https://github.com/intel/vtpm-td.

[72] Intel. Intel® TDX Module v1.5 TD Partitioning Architecture Specification March 2023, 2023. Last accessed on 2025-03-27. URL: https://www.intel.com/content/www/us/en/content-details/773039/intel-tdx-module-v1-5-td-partitioning-architecture-specification.html.

[73] Intel. Intel® Architecture Memory Encryption Technologies, 2024. Last accessed on 2025-03-27. URL: https://www.intel.com/content/dam/develop/external/us/en/documents-tps/multi-key-total-memory-encryption-spec.pdf.

[74] Intel. Intel® Trust Domain Extensions (Intel® TDX) Module Base Architecture Specification October 2024, 2024. Last accessed on 2025-03-27. URL: https://cdrdv2.intel.com/v1/dl/getContent/733575.

[75] Intel. Intel® Trust Domain Extensions, 2025. Last accessed on 2025-03-27. URL: https://cdrdv2-public.intel.com/690419/TDX-Whitepaper-February2022.pdf.

[76] Intel. Intel Software Guard Extensions, [n. d.]. Last accessed on 2025-03-27. URL: https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html.

[77] Intel. Intel® Trust Domain Extensions (Intel TDX), [n. d.]. Last accessed on 2025-03-27. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[78] iPerf Develoeprs. iPerf – The TCP, UDP and SCTP Network Bandwidth Measurement Tool, [n. d.]. Last accessed on 2025-03-27. URL: https://iperf.fr/.

[79] Deepak Sirone Jegan, Liang Wang, Siddhant Bhagat, and Michael Swift. Guarding serverless applications with kalium. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4087–4104, Anaheim, CA, August 2023. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/jegan.

[80] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. Association for Computing Machinery, 2021. doi:10.1145/3477132.3483541.

[81] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2021. doi:10.1145/3445814.3446701.

[82] Chao Jin, Zili Zhang, Xingyu Xiang, Songyun Zou, Gang Huang, Xuanzhe Liu, and Xin Jin. Ditto: Efficient Serverless Analytics with Elastic Parallelism. In *Proceedings of the ACM SIGCOMM 2023 Conference*. Association for Computing Machinery, 2023. doi:10.1145/3603269.3604816.

[83] The kernel development community. Nested VMX, [n. d.]. Last accessed on 2025-04-07. URL: https://docs.kernel.org/virt/kvm/x86/nested-vmx.html.

[84] Seong-Joong Kim, Myoungsung You, Byung Joon Kim, and Seungwon Shin. Cryonics: Trustworthy function-as-a-service using snapshot-based enclaves. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*. Association for Computing Machinery, 2023. doi:10.1145/3620678.3624789.

[85] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv—Optimizing the operating system for virtual machines. In *Proceedings of the 2014 USENIX Annual Technical Conference*. USENIX Association, 2014. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity.

[86] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating Function-as-a-Service Workflows. In *Proceedings of the 2021 USENIX Annual Technical Conference*. USENIX Association, 2021. URL: https://www.usenix.org/conference/atc21/presentation/kotni.

[87] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. Unikraft: Fast, specialized unikernels the easy way. In *Proceedings of the 16th European Conference on Computer Systems*. Association for Computing Machinery, 2021. doi:10.1145/3447786.3456248.

[88] Dmitrii Kuvaiskii, Dimitrios Stavrakakis, Kailun Qin, Cedric Xing, Pramod Bhatotia, and Mona Vij.

Gramine-tdx: A lightweight os kernel for confidential vms. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2024. doi:10.1145/3658644.3690323.

[89] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. From Zygote to Morula: Fortifying Weakened ASLR on Android. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. IEEE, 2014. doi:10.1109/SP.2014.34.

[90] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the 15th European Conference on Computer Systems*. Association for Computing Machinery, 2020. doi:10.1145/3342195.3387532.

[91] I.M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, 14(7), 1996. doi:10.1109/49.536480.

[92] Dingji Li, Zeyu Mi, Chenhui Ji, Yifan Tan, Binyu Zang, Haibing Guan, and Haibo Chen. Bifrost: Analysis and Optimization of Network I/O Tax in Confidential Virtual Machines. In *Proceedings of the 2023 USENIX Annual Technical Conference*. USENIX Association, 2023. URL: https://www.usenix.org/conference/atc23/presentation/li-dingji.

[93] Mengyuan Li, Yuheng Yang, Guoxing Chen, Mengjia Yan, and Yinqian Zhang. Sok: Understanding design choices and pitfalls of trusted execution environments. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*. Association for Computing Machinery, 2024. doi:10.1145/3634737.3644993.

[94] Mingyu Li, Yubin Xia, and Haibo Chen. Confidential serverless made efficient with plug-in enclaves. In *Proceedings of the ACM/IEEE 48th Annual International Symposium on Computer Architecture*. IEEE, 2021. doi:10.1109/ISCA52012.2021.00032.

[95] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *Proceedings of the 2022 USENIX Annual Technical Conference*. USENIX Association, 2022. URL: https://www.usenix.org/conference/atc22/presentation/li-zijun-rund.

[96] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing. In *Proceedings of the 2022 USENIX Annual Technical Conference*. USENIX Association, 2022. URL: https://www.usenix.org/conference/atc22/presentation/li-zijun-help.

[97] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. Glamdring: Automatic application partitioning for intel SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference*. USENIX Association, 2017. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind.

[98] Qingyuan Liu, Yanning Yang, Dong Du, Yubin Xia, Ping Zhang, Jia Feng, James R. Larus, and Haibo Chen. Harmonizing Efficiency and Practicability: Optimizing Resource Utilization in Serverless Computing with Jiagu. In *Proceedings of the 2024 USENIX Annual Technical Conference*, Santa Clara, CA, 2024. USENIX Association. URL: https://www.usenix.org/conference/atc24/presentation/liu-qingyuan.

[99] Costin Lupu, Andrei Albiundefinedoru, Radu Nichita, Doru-Florin Blânzeanu, Mihai Pogonaru, Răzvan Deaconescu, and Costin Raiciu. Nephele: Extending Virtualization Environments for Cloning Unikernel-based VMs. In *Proceedings of the 18th European Conference on Computer Systems*. Association for Computing Machinery, 2023. doi:10.1145/3552326.3587454.

[100] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2013. doi:10.1145/2451116.2451167.

[101] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *Proceedings of the 2021 USENIX Annual Technical Conference*. USENIX Association, 2021. URL: https://www.usenix.org/conference/atc21/presentation/mahgoub.

[102] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata,

Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*. Association for Computing Machinery, 2017. doi:10.1145/3132747.3132763.

[103] Ming Mao and Marty Humphrey. A performance study on the vm startup time in the cloud. In *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE, 2012. doi:10.1109/CLOUD.2012.103.

[104] Benshan Mei, Saisai Xia, Wenhao Wang, and Dongdai Lin. Cabin: Confining untrusted programs within confidential vms, 2024. URL: https://arxiv.org/abs/2407.12334, arXiv:2407.12334.

[105] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In *Proceedings of the 25th International Conference on Computer Aided Verification*. Springer-Verlag, 2013. doi:10.1007/978-3-642-39799-8_48.

[106] Jämes Ménétrey, Christian Göttel, Anum Khurshid, Marcelo Pasin, Pascal Felber, Valerio Schiavoni, and Shahid Raza. Attestation mechanisms for trusted execution environments demystified. In *Proceedings of the 2022 Distributed Applications and Interoperable Systems*. Springer International Publishing, 2022. doi:10.1007/978-3-031-16092-9_7.

[107] Microsoft. Azure Confidential Computing, 2019. Last accessed on 2025-03-27. URL: https://azure.microsoft.com/en-us/solutions/confidential-compute.

[108] Microsoft. Microsoft - Azure Functions, [n. d.]. Last accessed on 2025-03-27. URL: https://azure.microsoft.com/en-us/products/functions.

[109] Microsoft. Nested Virtualization in Azure, [n. d.]. Last accessed on 2025-04-04. URL: https://azure.microsoft.com/en-us/blog/nested-virtualization-in-azure/.

[110] Derek Mille. Arm CCA Planes and Interplane Communication Interface Proposal. Presented at 2024 Linux Plumbers Conference, 2024. Last accessed on 2025-03-27. URL: https://lpc.events/event/18/contributions/1864/.

[111] Masanori Misono, Peter Okelmann, Charalampos Mainas, and Pramod Bhatotia. uIO: Lightweight and Extensible Unikernels. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*. Association for Computing Machinery, 2024. doi:10.1145/3698038.3698518.

[112] 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), 2024. doi:10.1145/3700418.

[113] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. Twine: An embedded trusted runtime for webassembly. In *Proceedings of the IEEE 37th International Conference on Data Engineering*. IEEE, 2021. doi:10.1109/ICDE51399.2021.00025.

[114] Vikram Narayanan, Claudio Carvalho, Angelo Ruocco, Gheorghe Almasi, James Bottomley, Mengmei Ye, Tobin Feldman-Fitzthum, Daniele Buono, Hubertus Franke, and Anton Burtsev. Remote Attestation of Confidential VMs Using Ephemeral vTPMs. In *Proceedings of the 2023 Annual Computer Security Applications Conference*. Association for Computing Machinery, 2023. doi:10.1145/3627106.3627112.

[115] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. TrustZone Explained: Architectural Features and Use Cases. In *Proceedings of the 2nd International Conference on Collaboration and Internet Computing*. IEEE, 2016. doi:10.1109/CIC.2016.065.

[116] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with Serverless-Optimized containers. In *Proceedings of the 2018 USENIX Annual Technical Conference*. USENIX Association, 2018. URL: https://www.usenix.org/conference/atc18/presentation/oakes.

[117] OASIS Open. Virtual I/O Device (VIRTIO), 2023. Last accessed on 2025-03-27. URL: https://docs.oasis-open.org/virtio/virtio/v1.3/virtio-v1.3.html.

[118] Joongun Park, Naegyeong Kang, Taehoon Kim, Youngjin Kwon, and Jaehyuk Huh. Nested Enclave: Supporting Fine-grained Hierarchical Isolation with SGX. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. IEEE, 2020. doi:10.1109/ISCA45697.2020.00069.

[119] Joongun Park, Seunghyo Kang, Sanghyeon Lee, Taehoon Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Hardware-hardened sandbox enclaves for trusted serverless computing. *ACM Trans. Archit. Code Optim.*, 21(1), 2024. doi:10.1145/3632954.

[120] Federico Parola, Shixiong Qi, Anvaya B. Narappa, K. K. Ramakrishnan, and Fulvio Risso. SURE: Secure Unikernels Make Serverless Computing Rapid and

Efficient. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*. Association for Computing Machinery, 2024. doi:10.1145/3698038.3698558.

[121] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the Library OS from the Top Down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, page 291–304. Association for Computing Machinery, 2011. doi:10.1145/1950365.1950399.

[122] Shixiong Qi, Leslie Monis, Ziteng Zeng, Ian-chin Wang, and K. K. Ramakrishnan. SPRIGHT: Extracting the Server from Serverless Computing! High-performance eBPF-based Event-driven, Shared-memory Processing. In *Proceedings of the ACM SIGCOMM 2022 Conference*. Association for Computing Machinery, 2022. doi:10.1145/3544216.3544259.

[123] Weizhong Qiang, Zezhao Dong, and Hai Jin. Se-lambda: Securing privacy-sensitive serverless applications using SGX enclave. In *Proceedings of the 14th International Conference SecureComm*. Springer, 2018. doi:10.1007/978-3-030-01701-9_25.

[124] Ali Raza, Thomas Unger, Matthew Boyd, Eric B Munson, Parul Sohal, Ulrich Drepper, Richard Jones, Daniel Bristot De Oliveira, Larry Woodman, Renato Mancuso, Jonathan Appavoo, and Orran Krieger. Unikernel linux (ukl). In *Proceedings of the 18th European Conference on Computer Systems*. Association for Computing Machinery, 2023. doi:10.1145/3552326.3587458.

[125] AMD ASID related Github Issue. Minimum SEV non-ES ASID setting, 2025. Last accessed on 2025-03-27. URL: https://github.com/AMDESE/AMDSEV/issues/84.

[126] Yanjing Ren, Jingwei Li, Zuoru Yang, Patrick P. C. Lee, and Xiaosong Zhang. Accelerating Encrypted Deduplication via SGX. In *Proceedings of the 2021 USENIX Annual Technical Conference*. USENIX Association, 2021. URL: https://www.usenix.org/conference/atc21/presentation/ren-yanjing.

[127] Elena Reshetova and Carlos Bilbao. Confidential Computing in Linux for x86 virtualization, 2023. Last accessed on 2025-03-27. URL: https://docs.kernel.org/security/snp-tdx-threat-model.html.

[128] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2022. doi:10.1145/3503222.3507750.

[129] Wenwen Ruan, Wenhao Wang, Shuang Liu, Ran Duan, and Shoumeng Yan. Domainisolation: Lightweight intra-enclave isolation for confidential virtual machines. In *Science of Cyber Security*. Springer Nature Switzerland, 2023. doi:10.1007/978-3-031-45933-7_2.

[130] Marta Rybczyńska. Bounce Buffers for Untrusted Devices [LWN.net], 2019. Last accessed on 2025-03-27. URL: https://lwn.net/Articles/786558/.

[131] Andrea Sabbioni, Lorenzo Rosa, Armir Bujari, Luca Foschini, and Antonio Corradi. A Shared Memory Approach for Function Chaining in Serverless Platforms. In *Proceedings of the 2021 IEEE Symposium on Computers and Communications*. IEEE, 2021. doi:10.1109/ISCC53001.2021.9631385.

[132] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, et al. Xfaas: Hyperscale and low cost serverless functions at meta. In *Proceedings of the 29th Symposium on Operating Systems Principles*. Association for Computing Machinery, 2023. doi:10.1145/3600006.3613155.

[133] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using arm trustzone to build a trusted language runtime for mobile applications. *SIGARCH Comput. Archit. News*, 42(1):67–80, February 2014. doi:10.1145/2654822.2541949.

[134] Vasily A. Sartakov, Daniel O'Keeffe, David Eyers, Lluís Vilanova, and Peter Pietzuch. Spons & shields: practical isolation for trusted execution. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. Association for Computing Machinery, 2021. doi:10.1145/3453933.3454024.

[135] Vasily A. Sartakov, Lluís Vilanova, David Eyers, Takahiro Shinagawa, and Peter Pietzuch. CAP-VMs: Capability-Based Isolation and Sharing in the Cloud. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2022. URL: https://www.usenix.org/conference/osdi22/presentation/sartakov.

[136] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. Cubicleos: a library os with software componentisation for practical isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 2021. doi:10.1145/3445814.3446731.

[137] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. EbbRT: A framework for building Per-Application library operating systems. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2016. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/schatzberg.

[138] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. WESEE: Using Malicious #VC Interrupts to Break AMD SEV-SNP. In *Proceedings of the 45rd IEEE Symposium on Security and Privacy*. IEEE, 2024. doi:10.1109/SP54263.2024.00262.

[139] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. HECKLER: Breaking Confidential VMs with Malicious Interrupts. In *Proceedings of the 33rd USENIX Security Symposium*. USENIX Association, 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/schl%C3%BCter.

[140] Carlos Segarra, Tobin Feldman-Fitzthum, Daniele Buono, and Peter Pietzuch. Serverless confidential containers: Challenges and opportunities. In *Proceedings of the 2nd Workshop on SErverless Systems, Applications and MEthodologies*. Association for Computing Machinery, 2024. doi:10.1145/3642977.3652097.

[141] Amazon Web Servies. AWS Lambda - Run Code without Thinking about Servers or Clusters, [n. d.]. Last accessed on 2025-03-27. URL: https://aws.amazon.com/lambda/.

[142] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of the 2020 USENIX Annual Technical Conference*. USENIX Association, 2020. URL: https://www.usenix.org/conference/atc20/presentation/shahrad.

[143] Jiacheng Shi, Jinyu Gu, Yubin Xia, and Haibo Chen. Serverless Functions Made Confidential and Efficient with Split Containers. In *Proceedings of the 34th USENIX Security Symposium*. USENIX Association, 2025. URL: https://www.usenix.org/conference/usenixsecurity25/presentation/shi-jiacheng.

[144] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *Proceedings of the 2020 USENIX Annual Technical Conference*. USENIX Association, 2020. URL: https://www.usenix.org/conference/atc20/presentation/shillaker.

[145] Dimitrios Stavrakakis, Dimitra Giantsidi, Maurice Bailleu, Philip Sändig, Shady Issa, and Pramod Bhatotia. Anchor: A Library for Building Secure Persistent Memory Systems. *Proc. ACM Manag. Data*, 1(4), December 2023. doi:10.1145/3626718.

[146] Ariel Szekely, Adam Belay, Robert Morris, and M. Frans Kaashoek. Unifying serverless and microservice workloads with sigmaos. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. Association for Computing Machinery, 2024. doi:10.1145/3694715.3695947.

[147] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. Sequoia: Enabling Quality-of-service in Serverless Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. Association for Computing Machinery, 2020. doi:10.1145/3419111.3421306.

[148] Confidential Containers Dev Team. Confidential Containers, 2025. Last accessed on 2025-04-03. URL: https://github.com/confidential-containers/confidential-containers?tab=readme-ov-file.

[149] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Cntr: Lightweight OS Containers. In *Proceedings of the 2018 USENIX Annual Technical Conference*. USENIX Association, 2018. URL: https://www.usenix.org/conference/atc18/presentation/thalheim.

[150] Jörg Thalheim, Harshavardhan Unnibhavi, Christian Priebe, Pramod Bhatotia, and Peter Pietzuch. rkt-io: A Direct I/O Stack for Shielded Execution. In *Proceedings of the 16th European Conference on Computer Systems*. Association for Computing Machinery, 2021. URL: https://doi.org/10.1145/3447786.3456255.

[151] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. ShieldBox: Secure Middleboxes using Shielded Execution. In *Proceedings of the 2018 Symposium on SDN Research*. Association for Computing Machinery, 2018. doi:10.1145/3185467.3185469.

[152] Bohdan Trach, Oleksii Oleksenko, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. Clemmys: Towards Secure Remote Execution in FaaS. In *Proceedings of the 12th ACM International Conference on Systems and Storage*. Association for Computing Machinery, 2019. doi:10.1145/3319647.3325835.

[153] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and Security Isolation of Library OSes for Multi-process Applications. In *Proceedings of the 9th European Conference on Computer Systems*. Association for Computing Machinery, 2014. doi:10.1145/2592798.2592812.

[154] Dmitrii Ustiugov, Dohyun Park, Lazar Cvetković, Mihajlo Djokic, Hongyu He, Boris Grot, and Ana Klimovic. Enabling In-Vitro Serverless Systems Research. In *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless*. Association for Computing Machinery, 2023. doi:10.1145/3605181.3626191.

[155] Thomas Van Strydonck, Job Noorman, Jennifer Jackson, Leonardo Alves Dias, Robin Vanderstraeten, David Oswald, Frank Piessens, and Dominique Devriese. Cheri-tree: Flexible enclaves on capability machines. In *Proceedings of the IEEE 8th European Symposium on Security and Privacy*. IEEE, 2023. doi:10.1109/EuroSP57164.2023.00070.

[156] Jian Wang, Kwame-Lante Wright, and Kartik Gopalan. Xenloop: a transparent high performance inter-vm network loopback. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*. Association for Computing Machinery, 2008. doi:10.1145/1383422.1383437.

[157] Wenhao Wang, Linke Song, Benshan Mei, Shuang Liu, Shijun Zhao, Shoumeng Yan, XiaoFeng Wang, Dan Meng, and Rui Hou. The Road to Trust: Building Enclaves within Confidential VMs. In *Proceedings of the 32nd Annual Network and Distributed System Security Symposium*. Internet Society, 2025. doi:10.14722/ndss.2025.240385.

[158] Ofir Weisse, Valeria Bertacco, and Todd Austin. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. Association for Computing Machinery, 2017. doi:10.1145/3079856.3080208.

[159] Yuanchao Xu, James Pangia, Chencheng Ye, Yan Solihin, and Xipeng Shen. Data enclave: A data-centric trusted execution environment. In *Proceedings of the 2024 IEEE International Symposium on High-Performance Computer Architecture*. IEEE, 2024. doi:10.1109/HPCA57654.2024.00026.

[160] Jason Zhijingcheng Yu, Conrad Watt, Aditya Badole, Trevor E. Carlson, and Prateek Saxena. Capstone: A capability-based foundation for trustless secure memory access. In *Proceedings of the 32nd USENIX Security Symposium*. USENIX Association, 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/yu-jason.

[161] Minchen Yu, Tingjia Cao, Wei Wang, and Ruichuan Chen. Following the data, not the function: Rethinking function orchestration in serverless computing. In *Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2023. URL: https://www.usenix.org/conference/nsdi23/presentation/yu.

[162] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 2020 ACM Symposium on Cloud Computing*. Association for Computing Machinery, 2020. doi:10.1145/3419111.3421280.

[163] Chuqi Zhang, Rahul Priolkar, Yuancheng Jiang, Yuan Xiao, Mona Vij, Zhenkai Liang, and Adil Ahmad. Erebor: A Drop-In Sandbox Solution for Private Data Processing in Untrusted Confidential Virtual Machines. In *Proceedings of the 20th European Conference on Computer Systems*. Association for Computing Machinery, 2025. doi:10.1145/3689031.3717464.

[164] Fengzhe Zhang, Jin Chen, Haibo Chen, and Binyu Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-Tenant Cloud with Nested Virtualization. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, 2011. URL: https://doi.org/10.1145/2043556.2043576.

[165] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. Association for Computing Machinery, 2021. doi:10.1145/3477132.3483580.

[166] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. KylinX: A dynamic library operating system for simplified and efficient cloud virtualization. In *Proceedings of the 2018 USENIX Annual Technical Conference*. USENIX Association, 2018. URL: https://www.usenix.org/conference/atc18/presentation/zhang-yiming.

[167] Shixuan Zhao, Pinshen Xu, Guoxing Chen, Mengya Zhang, Yinqian Zhang, and Zhiqiang Lin. Reusable Enclaves for Confidential Serverless Computing. In *Proceedings of the 32nd USENIX Security Symposium*. USENIX Association, 2023. URL:

https://www.usenix.org/conference/usenixsecurity23/presentation/zhao-shixuan.

[168] Yuxuan Zhao, Weikang Weng, Rob van Nieuwpoort, and Alexandru Uta. In serverless, os scheduler choice costs money: A hybrid scheduling approach for cheaper faas. In *Proceedings of the 25th International Middleware Conference*. Association for Computing Machinery, 2024. doi:10.1145/3652892.3700757.

[169] Ziqiao Zhou, Anjali, Weiteng Chen, Sishuai Gong, Chris Hawblitzel, and Weidong Cui. VeriSMo: A verified security module for confidential VMs. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2024. URL: https://www.usenix.org/conference/osdi24/presentation/zhou.

Appendix

A Detailed Security Analysis

Guest OS. First, we consider attacks originating from the untrusted guest OS within the CVM. An attacker may attempt to access data of a serverless request. However, this data is encrypted with a user key that only the trusted monitor possesses and is inaccessible to external entities, provided that the user secrets are not compromised. On top of that, modifying the VMPL configuration via the RMPADJUST instruction [14] is prohibited in the guest OS (VMPL-2). Thus, any attempt from the guest OS to access memory regions of the trusted monitor and Trusted Processes fails thanks to VMPL protections, which are set by the trusted monitor and trap such accesses.

Further, an attacker could load arbitrary zygotes and functions. However, Wallet's attestation service blocks this process if there is a measurement discrepancy from the values in the function provider's policy. Replay attacks by replacing user input are also detected as users can validate the integrity of the input based on the content of the attestation report. Lastly, if an attacker intercepts a file request from a trustlet to the nested filesystem (§ 5.2) and returns a compromised file, the LibOS verifies the file's measurement against its expected value in Wallet's manifest, rejecting any tampered files.

Host and hypervisor. CVMs employ mechanisms to protect CVM's private memory and provide ways to retrieve trusted system information from within the CVM. Precisely, unauthorized access to CVM memory is blocked by AMD SEV-SNP's reverse map page table (RMP), which prevents the host, hypervisor, and other co-located VMs from accessing CVM's private memory. Direct memory access (DMA) attacks from the host are also infeasible since DMA is prohibited by hardware. Further, to address potential threats to compromise the guest via arbitrary interrupt or exception injection [138, 139], SEV-SNP introduces alternate interrupt injection [9], restricting the hypervisor to specific exception types. On top of that, an attacker may try to disrupt the system by tampering with the CPUID instruction [93] return values. As a countermeasure, SEV-SNP provides a CPUID-page mechanism, which ensures correct CPUID values, validated by the ASP.

Attempts to compromise Wallet by launching a tampered trusted monitor, bypassing it altogether, or directly booting Wallet in a standard VM are detected during Wallet's remote attestation workflow. Attestation reports include invalid measurement values if altered software is used. Attempts to deploy Wallet as a normal VM can be identified, as, in this case, Wallet fails to produce a valid report from the ASP. Even crafted reports are detected, since only the CPU vendor's key, securely managed by the ASP, can sign genuine attestation certificates. Lastly, establishing secure connections (e.g., TLS) protects against network communication manipulation by ensuring the confidentiality and integrity of network packets.

Co-located serverless functions. Wallet prevents attacks from co-located functions. An attacker might create a serverless function that exploits a runtime vulnerability aiming to access memory belonging to another trustlet. However, each trustlet runs in ring3 and has its own page tables managed by the trusted monitor, which prohibits access to other trustlet's memory. Further, to prevent reuse-based attacks [167], Wallet's process runtime recreates trustlets when receiving a request from a different user, resetting the potentially compromised state due to a previous invocation.

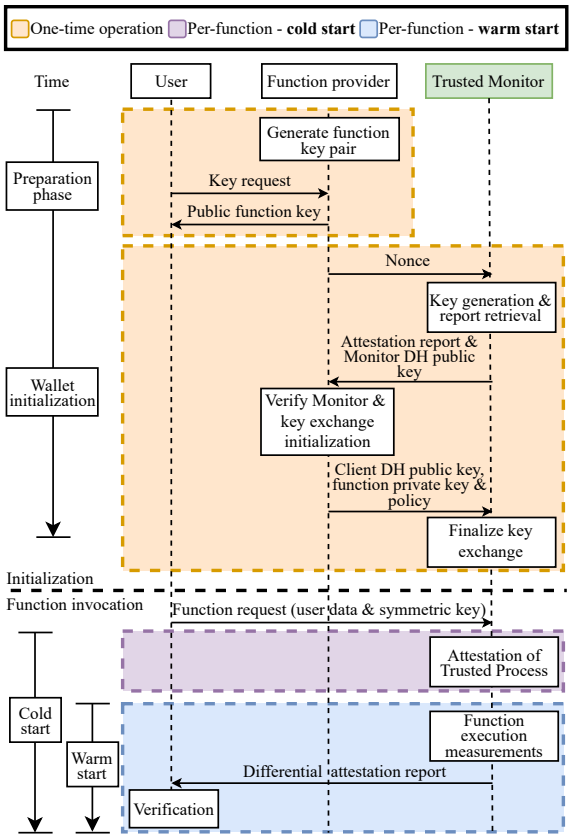

Figure 11: Workflow of Wallet's differential attestation.

B Differential Attestation

B.1 Detailed Attestation Flow

In confidential serverless computing, besides attesting the underlying platform, users need to verify the serverless runtime, the deployed function, and any inputs and outputs to validate the correct execution. This process can severely impact the function's startup and execution latency. Wallet's *differential attestation protocol* alleviates this issue. The key idea is to retain previously calculated measurements and compose a cumulative attestation report for the function execution.

Attestation flow. Figure 11 illustrates the workflow of Wallet's differential attestation protocol. It consists of two main phases: the *initialization* and the *function invocation* phase.

The **initialization** phase begins with the function provider sharing the public function key with the user, which is used to encrypt the user requests. Then, during the Wallet initialization phase, the function provider establishes a secure connection with the Wallet instance to share the function's private

key and the function's policy that specifies the measurements of zygotes and functions permitted for use. The establishment of the secure connection relies on the attestation of the trusted monitor, whose trust is rooted in the ASP [14].

Precisely, the function provider first sends a nonce to the trusted monitor to prevent replay attacks. In turn, the attestation service generates a Diffie–Hellman (DH) key pair and retrieves the attestation report from the ASP, which includes the measurement of the trusted monitor, the provided nonce and the hash value of the DH public key as user data. The function provider receives and verifies the report and then sends the function private key and the policy to the Wallet instance. Wallet uses the function's private key to decrypt user requests and the policy to attest the trusted processes.

The **function invocation** phase includes two scenarios depending on the Wallet's state, namely *cold* and *warm* start. In the cold start case, the attestation service verifies the integrity of the trusted process by comparing the measurements of its components against the values provided by the function provider's policy during the initialization phase. Beyond this point, subsequent invocations of the same function become a warm start. Wallet's attestation service only has to measure the function input and output and can generate a report based on prior measurements of the monitor, the zygote, and the invoked function. When several serverless functions are chained, the extended attestation report includes all their measurements. This approach reduces the attestation time while ensuring verifiability for the entire data chain.

B.2 Formal Verification of Differential Attestation

We formally verify Wallet's differential attestation protocol, as depicted in Figure 11, using the Tamarin Prover. We model the steps specific to Wallet in detail.

More precisely, an unbounded number of protocol flows are considered simultaneously, with their individual states represented as part of a global multiset. State transitions are encoded as rewriting rules, which are one of the input formats for the Tamarin prover, and long-term secrets are treated as compromisable. Messages are considered atomic, and cryptographic functions (e.g., hashing, encryption) are assumed to be perfect, i.e., without side effects or collisions, and known to the attacker. We further assume the correct functionality of the underlying hardware, i.e., the ASP does not leak its secrets and produces correct reports, and report verification infrastructure. Such external dependencies are treated as a black box.

Our model is based on *action facts*, which denote events in the protocol trace. We use the *Secret(s)* action fact to explicitly mark secret information *s*, in conjunction with the builtin *K(x)* action fact, which marks an adversary obtaining information *x*. We further introduce the *UsrTrustRes(...)* action fact to indicate that a user trusts, after the last protocol step, that the obtained result was indeed computed on a trusted software stack.

Rules. To translate the protocol specification to rules, which operate on a global state, we identify: *(i)* the necessary inputs from the network, as well as, the persistent states for each Agent for the rules left-hand side, *(ii)* the resulting outputs on the network, as well as, any modifications to the persistent states of each Agent for the rules right-hand side, *(iii)* any checks performed by an Agent, which may translate to restrictions of the rule transitions.

This approach enables a systematic derivation of the model from the specification for most of the protocol, some parts, however, require additional consideration:

The preparation phase contains an exchange of the public function key, between the user and the function provider, which is not further specified, for the formal analysis we model this exchange as a secure channel (Figure 12). The motivation for this is, that we do not consider attacks on this information exchange. The attacker is still able to obtain the function's public key, which we model with *Out(pk(~func_priv))* on the right-hand side of the key generation rule, but unable to provide the user false information in the preparation phase.

```
rule ChanOut_S:
   [ Out_S($A,$B,x) ]
   --[ ChanOut_S($A,$B,x) ]->
   [ !Sec($A,$B,x) ]

rule ChanIn_S:
   [ !Sec($A,$B,x) ]
   --[ ChanIn_S($A,$B,x) ]->
   [ In_S($A,$B,x) ]
```

Figure 12: Secure channel rules from the Tamarin Manual [49].

The initialization phase is largely based on Tamarin's *diffie-hellman* builtin, for modelling the key exchange. To model the report generation, verification and user-data retrieval as a black-box we introduce the functions *gen*, *verif* and *getD* respectively. The expected behaviour of these is expressed in the following equations, where m refers to the machine, d to the software measurement, and u to the retrievable user-data:

$$verif(gen(m,d,u),m,d) = true \tag{1}$$

$$getD(gen(m,d,u)) = u \tag{2}$$

In this phase we also included an unrestricted *machine_init* to be able to set up an infinite number of machines and thus infinite potential Wallet instances. This, together with the way we modeled the function provider in the previous phase, allows us to model an unbounded number of Wallet instances, machines, functions, providers, and users all while ensuring they are unable to interfere to violate our desired properties.

The remainder of the rules are straightforward translations from the protocol.

Lemmas. We use a @ t_i to denote that action fact a occurred at time t_i. Leveraging these action facts, we verify the following lemmas for Wallet's differential attestation protocol:

- **Secrecy lemma:** If some data (e.g., function input/output) is declared as secret *s*, it remains undisclosed to attackers unless one agent *A* having access to it (*KnowSecret(A, s)*) is

Table 5: Agents with access to secret information

Information	Agents with access
User's Function input	User, TrustedMonitor, Function Provider
Output encryption key	User, TrustedMonitor, Function Provider
Function's private key	TrustedMonitor, Function Provider

explicitly compromised (*Compr(A)*).

$$\forall\, s,t_i \,.\, Secret(s)@t_i \implies \nexists\, t_j \,.\, K(s)@t_j \vee (\exists\, A,t_r \,.\, Compr(A)@t_r \wedge KnowSecret(A,\, s)@t_i) \quad (3)$$

• **Authenticity lemma:** If a user trusts a result, after verifying the differential attestation report, then the result *res* was computed on the trusted monitor in a valid state *tm*, and with the expected zygote *zy*, trustlet *tr*, and input *in*, which is modeled by the $compute$ function.

$$\forall\, tm,zy,tr,in,res,t_i \,.\, UsrTrustRes(tm,zy,tr,in,res)@t_i \implies compute(tm,zy,tr,in) = res \quad (4)$$

Tamarin verifies the specified properties by showing that there is no trace that leads to a falsification of these lemmas. Thus, we show that the attestation protocol ensures secrecy, authenticity, and correctness under the specified assumptions.

However, to increase confidence in the correctness of our model we include a simple sanity check, which ensures that it is possible to reach this point in the protocol. For this we use this lemma:

$$\exists\, tm,zy,tr,in,res,t_i \,.\, UsrTrustRes(tm,zy,tr,in,res)@t_i \quad (5)$$

If this lemma is falsified, then the *authenticity lemma* would trivially hold, since it would be a for-all condition on an empty set. However, this would likely indicate an issue in our model and not the proper functioning of the protocol, therefore we include the above sanity check to rule out this edge case.

For the secrecy analysis, we were also required to explicitly state the agents that can potentially access, i.e., obtain in plain text, certain secret information, as depicted in Table 5. We found the sets of agents to be minimal, as further reduction caused the *secrecy lemma* to be violated. Notably, this required us to include the *Function Provider* as an agent, with potential access to the User's function input and output. The reason for this is the private function key, which is known to the function provider.

The persistence of these long-term keys and their use also prevented us from verifying a stronger secrecy property called **perfect forward secrecy**. This would ensure that information will remain secret, even if the attacker is able to record the communication and *compromize* some long-term key in the future. Checking for this property entails making a small modification to the *secrecy lemma*:

$$\forall\, s,t_i \,.\, Secret(s)@t_i \implies \nexists\, t_j \,.\, K(s)@t_j \vee (\exists\, A,t_r \,.\, Compr(A)@t_r \wedge KnowSecret(A,\, s)@t_i \wedge \boldsymbol{t_r < t_i}) \quad (6)$$

For our protocol, Tamarin is able to correctly identify that this lemma is violated since compromising the long-term private key of a function enables the decryption of any previous request to that function. We consider this attack vector out of scope for this paper, as it relies on *compromising* a key component of our infrastructure. Future work may explore extensions to the protocol, that address this, to better contain an initial compromise.

C WALLET Implementation Details

Trusted monitor. We implement our prototype of WALLET for Linux environments. We base WALLET's trusted monitor on COCONUT-SVSM [38]. COCONUT-SVSM is a service module that operates at VMPL-0 and aims to provide services (e.g., vTPM [114]) to a guest running at VMPL-2. We extend COCONUT-SVSM to support trusted processes in VMPL-1 and incorporate WALLET's differential attestation service. We base our LibOS for trusted processes on Gramine [27, 39] and implement a backend to enable its execution at VMPL-1. WALLET's current prototype uses a patched Linux version v6.8 (host OS) and v6.5 (guest OS), configured to support COCONUT-SVSM.

Communication protocol. The COCONUT-SVSM implements a guest communication protocol defined by AMD [12], enabling the guest OS to request services from the COCONUT-SVSM using the VMGEXIT instruction. We extend this protocol to support the monitor system calls (Table 2) by adding a new service type and implementing its corresponding handlers.

For communication between the trusted monitor and trustlets, we enable #VC (VMM Communication Exception) reflection [14], allowing the trusted monitor to trap #VC during trustlet execution. We define a communication protocol based on the cpuid instruction, leveraging the unused hypervisor CPUID Leaf range [13]. Precisely, trustlets set a service type in the RAX register and execute the cpuid instruction, triggering #VC. In turn, the trusted monitor inspects the RAX value, executes the corresponding monitor service, if applicable, or the normal cpuid instruction, and returns the result.

LibOS. Our LibOS for trusted processes uses Gramine [27, 39] with a backend for VMPL-1 execution. Gramine defines a Platform Adaptation Layer (PAL) host ABI [40] to delegate specific operations to the host. We implement our PAL as a shim layer for our trusted monitor, which allows WALLET to run the Python runtime as a trusted process. Its core functionalities include memory allocation and external file access, and it leverages WALLET's communication protocol to perform its PAL ABI operations requests.

For memory allocations, the trusted monitor handles the requests from the PAL. Specifically, it allocates memory, adjusts the VMPL access permissions using the RMPADJUST instruction, and updates the trustlet's page table entries. For external file access, the trusted monitor returns a special value to the guest OS to request the desired file. Then, the guest OS reads the file and invokes the invoketrustlet() monitor call. Subsequently, the trusted monitor copies the file data into the trustlet's memory and resumes its execution. The trustlet validates the file integrity before using it by comparing its

measurement with a specified value in the manifest. For the memory backend, the trusted monitor secures the memory pool and validates each page during its boot to optimize memory allocation at runtime. When a trustlet requests memory, the trusted monitor allocates memory from the pool, and updates the trustlet's page table entries accordingly.

Copy-on-write handling. Gramine LibOS operates in ring3 and does not contain exception handling code. To realize copy-on-write-based forking, we develop a minimal exception handler that executes in ring0 within a trustlet (VMPL-1). The trusted monitor configures the GDT (Global Descriptor Table), IDT (Interrupt Descriptor Table), and TSS (Task State Segment) of the trustlet to ensure that it uses the handler upon launch. On a page fault, the page fault handler delegates the actual handling to the trusted monitor via a monitor call. The trusted monitor resolves it by updating the page table entries appropriately and then resumes the execution of the trustlet.

Attestation service. Wallet leverages COCONUT-SVSM's `get_regular_report()` to retrieve a signed AMD SEV-SNP report and, then, stores it securely in the trusted monitor's memory. To calculate the measurements of individual trusted processes components (e.g., zygote image, trustlet), Wallet uses the SHA-512 algorithm (same as the SEV-SNP's measurement algorithm [14]). These measurements are cached in the zygote and trustlet contexts for faster retrieval. The monitor invokes the attestation functions internally when fetching a zygote or creating a trustlet, validating them against the function provider's policy by comparing the calculated with the provided measurements. Finally, on a function invocation, the measurements of the used software components and the function input and output are appended to the trusted monitor's attestation report, which is then signed using the private key provided by the function provider and forwarded to the user for verification of the entire execution.

Guest OS components. Wallet includes a kernel module that allows a guest application to make monitor calls. In addition, we implement a Python library that interacts with the trusted monitor through this kernel module. We also extend the SeBS benchmark suite [34] to support running serverless functions with Wallet using this Python library.