Serverless Functions Made Confidential and Efficient with Split Containers

Jiacheng Shi, Jinyu Gu, Yubin Xia, Haibo Chen

Institute of Parallel and Distributed Systems (IPADS), SEIEE, Shanghai Jiao Tong University
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

Abstract

The increasing adoption of serverless computing in security-critical fields (e.g., finance and healthcare) motivates confidential serverless. This paper explores confidential virtual machines (CVMs), a promising hardware security feature offered by various CPU architectures, for securing serverless functions. However, our analysis reveals a mismatch between current CVM implementations and function needs, resulting in performance bottlenecks, resource inefficiency, and an expanded trusted computing base (TCB).

We present *split container*, a design that creates confidential containers with a minimal TCB. Our observation is that real-world serverless functions often require a limited set of OS functionalities. Thus, our design deploys a function-oriented OS (microkernel + library OS) within the CVM for secure execution of multiple functions while reusing an untrusted commodity OS like Linux outside for container management. Based on the split container design, we have implemented CoFunc, a system prototype that works on both AMD SEV and Intel TDX. With FunctionBench and ServerlessBench, CoFunc demonstrates significant performance improvements (up to 60× on SEV and 215× on TDX) compared to the only known CVM-based confidential container (Kata-CVM with optimizations), while incurring <14% performance overhead on average compared to a state-of-the-art non-confidential container system (lean container).

1 Introduction

Serverless computing has emerged as a popular computing paradigm over the past decade [46, 102, 65, 41, 72, 75]. The Function-as-a-Service (FaaS) paradigm allows users to concentrate on the business logic using functions, and offers a number of benefits, including elasticity, cost-effectiveness, and automated deployment and management. Owing to such virtues, it has been widely adopted in various application scenarios, including security-critical or privacy-sensitive ones, such as facial recognition [101, 64, 62], healthcare [85], finance [54, 76], and many others.

Existing serverless platforms [44, 59, 68, 38, 79] run functions in containers or virtual machines (VMs). This implies that the security of a function depends on the entire software stack, including OS, hypervisor, and management tools, which comprises a huge TCB. To this end, confidential containers [45] have been introduced by combining trusted execution environment (TEE) and containers to provide both security and ease of deployment, which have been increasingly adopted by the industry in recent years [9, 32, 11, 25, 10]. Among these systems, CVM-based confidential containers, like Kata-CVM [23], are getting popular and have been increasingly adopted by cloud providers [11, 25].

Unfortunately, existing CVM-based confidential container systems are not suitable for serverless workloads, due to the dilemma of boot latency and resource provisioning. On one hand, the ephemeral and auto-scaling nature of functions requires instant and concurrent launching. Specifically, 50% of real-world serverless functions execute for less than 1s [111], and one server may receive 200 function launching requests simultaneously [93, 94]. However, Kata-CVM runs one container per function within one CVM, and booting 200 SEV microVMs [70] concurrently takes 28.5s due to the serial measurement of each CVM (Figure 3). On the other hand, although keeping warm CVMs for functions could mitigate cold start, it incurs severe resource provisioning problem since memory sharing is not allowed between CVMs.

In this paper, we explore a sharing-CVM design for serverless which allows multiple functions (confidential containers) to run within the same CVM. It avoids CVM cold boots on the critical path and enables memory sharing between function instances. However, this design faces security challenges because the inter-function isolation relies on the CVM guest OS. A commodity OS like Linux is large and complex and increases the TCB as well as attack surfaces. Two insights lead to addressing this concern: (i) Container management, such as resource allocation and namespace provisioning, can be decoupled from function execution, and hence a large portion of code can be deployed in the host Linux outside the CVM. (ii) The serverless programming model (e.g., stateless, simple interaction, platform parallelism instead of multi-processing/threading) [111, 80, 65, 5, 125, 79, 104, 60] determines that function execution typically relies on a fraction of system calls (74 in our evaluation). As a result, it is feasible to use a small kernel to serve as the CVM guest OS.

We propose **Co**nfidential **Func**tions (CoFunc for short) that decouples container security from management, through a *split container* architecture: each confidential container inside the CVM is paired with a *shadow container* outside the CVM (see Figure 1 rightmost). A shadow container is a normal Linux container with cgroups [6] and namespaces [27] used for management. It assigns its CPU and memory resources to its corresponding confidential container (*resource*

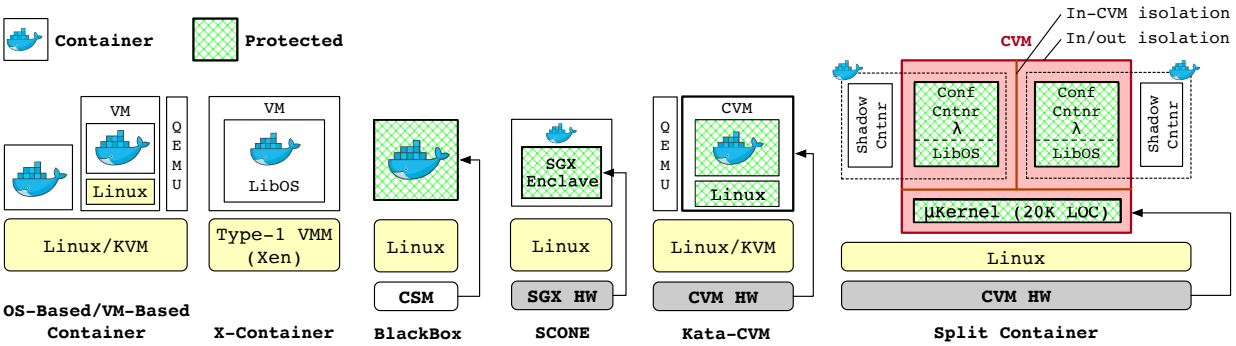

Figure 1. Comparison of different container architectures. Conf: confidential, Cntnr: container.

granting) in which the function executes. Within the CVM, we deploy a function-oriented OS (microkernel + library OS) to minimize the kernel-mode TCB while offering the system calls (syscall) needed by functions. A CVM microkernel (20K lines of code) runs in the CVM kernel mode and manages the isolation of confidential containers. Each confidential container has an individual function library OS (libOS) running in the user mode of its own address space. We also reuse the I/O functionalities (i.e., container rootfs and network) of the host Linux to avoid introducing complex I/O stacks and drivers into the CVM. Since the host Linux is untrusted, the libOS uses encryption and authentication for I/O protection.

CoFunc achieves the security goal analogous to prior confidential containers [45, 69], namely, protecting the confidentiality and integrity of a confidential container from the compromised privileged software like Linux outside the CVM, malicious co-located confidential containers inside the CVM, and even physical attacks. We implement a prototype of CoFunc that can run on both AMD SEV and Intel TDX servers, and evaluate it with real-world functions and benchmarks [55, 81, 125, 90]. Compared with lean container [103, 121], a native non-confidential Linux container optimized for serverless, CoFunc incurs 2.7%∼26.7% (TDX) and 0.7%∼17.3% (SEV) overhead. Compared with vanilla Kata-CVM [22] on TDX and our optimized Kata-CVM with specialized SEV microVM [70], CoFunc shows an average 44× (1.06∼215×) and 12× (1.02∼60×) speedup in end-to-end latency, and requires 20∼56× less memory when booting 200 functions.

In summary, this paper makes the following contributions:

- *CVM+Serverless*: As far as we known, there was no prior work that explores the integration of CVMs with serverless computing when our research started. We present the first comprehensive design space exploration that reveals a mismatch between them.
- *Split-Container*: Instead of implementing container management functionalities within the CVM, we propose a split container design that transparently passes through containerization offered by the host into the CVM. This is different from all existing secure/confidential container architectures.
- *CVM-OS*: We propose a new CVM OS that revives the exokernel concept for CVMs, being the first non-Linux CVM OS prototyped on both SEV and TDX. We have also made it open-source.
- *Demonstration*: We build a system prototype named CoFunc on both Intel TDX and AMD SEV platforms, with experimental results on real-world functions demonstrating its effectiveness and efficiency.

2 Motivation

With the growing adoption of serverless computing in areas that handle sensitive data, such as finance [54, 76], healthcare [85], and facial recognition [101], security has become a major concern [33, 62]. This section revisits the typical container designs (see Figure 1) and explores how CVM can be leveraged to build confidential containers for running serverless functions (as illustrated in Figure 2a).

2.1 Revisiting Existing Container Architectures

OS-based containers [115, 24, 30] have been criticized for poor isolation because of sharing a large kernel like Linux [123, 95, 117]. A malicious container can attack other containers after compromising the shared kernel. VM-based containers [22, 12, 29, 38, 100] have been proposed to improve the isolation by running each container inside one VM. A VM-based container cannot attack others after compromising its own guest OS. Yet, containers can still attack the host OS via privilege escalation by compromising QEMU/KVM [15, 37, 14].

X-Containers [112] leverages Type-1 virtualization (i.e., Xen [47]) to reduce the TCB to improve container isolation. However, it faces compatibility challenges as it cannot be deployed in the prevalent Type-2 virtualization environments used in cloud computing, such as Linux/KVM [107].

BlackBox [69] employs a container security monitor (CSM) with a small TCB in the hypervisor privilege level to protect containers on untrusted Linux. CSM plays a role in intercepting the system calls issued by containers, which may cause moderate performance overhead. Meanwhile, it

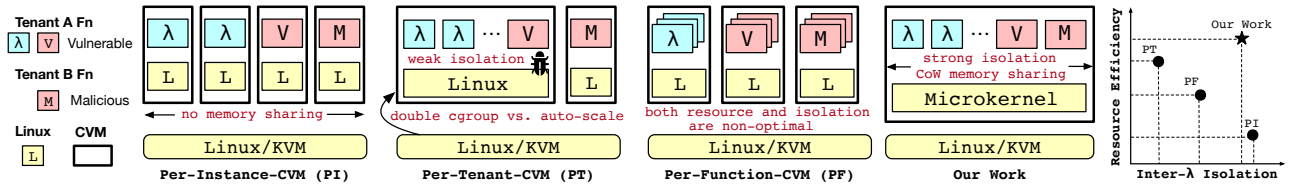

Figure 2. An exploration on the design space of CVM-based confidential containers for functions.

is incompatible with the existing VM virtualization stack because the hypervisor privilege level is occupied by CSM.

SCONE [45] uses Intel SGX enclaves to protect containers, which removes Linux from the TCB and achieves high security such as defending against physical attacks [124, 67]. However, SGX is not available on other CPU architectures, and the slow startup of SGX enclaves does not fit ephemeral serverless functions. PIE [90] allows sharing of secure memory among enclaves to accelerate the creation process of an enclaved function, which requires non-trivial hardware modifications and hence is not commercially available.

2.2 Design Space: CVM-based Confidential Containers

TEEs have proven to be effective in improving the security of outsourced computation [45, 96, 74]. In recent years, various CPU vendors have introduced CVMs, including AMD SEV [1], Intel TDX [21], ARM CCA [4], OpenPOWER PEF [73], etc. These CVM implementations provide a consistent abstraction, safeguarding an entire VM from external threats, including compromised hypervisors and physical attacks. This section explores the design space of CVM-based confidential containers for serverless functions.

Design-choice-1: Per-Instance-CVM. One design choice is to replace each VM with a CVM for VM-based containers, as adopted by the state-of-the-art, Kata-CVM [22, 23]. Specifically, Kata-CVM launches a separate CVM for each function instance, ensuring strong isolation through hardware protection. This prevents a compromised host OS (Linux/KVM) from inspecting or tampering with the container's memory or execution. However, this design faces the dilemma of cold start and resource provisioning.

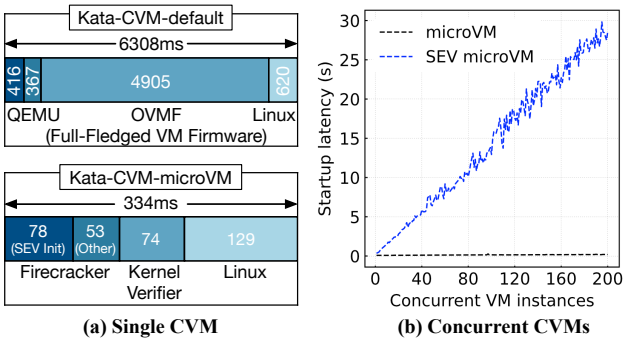

Figure 3. (a) Breakdown of the startup latency of a default SEV CVM or an optimized SEV microVM, (b) the end-to-end startup latency of concurrent SEV microVMs.

The problem of cold start. The boot latencies of one CVM and concurrent CVMs (shown in Figure 3) are too high to be acceptable for ephemeral functions. Booting a default Kata-CVM takes 6.3s on AMD SEV. After applying the microVM optimization [70], the latency is still 334ms, while 50% of real-world serverless functions execute for less than 1s [111]. Meanwhile, the end-to-end startup latency increases linearly with the number of concurrent SEV microVMs. The reason why multi-core does not help is that there exists a single-core Platform Security Processor which manages the SEV CVM launch commands sequentially [35]. Yet, the autoscaling of serverless computing makes that one cloud machine can receive 200 simultaneous function requests [93]. The end-to-end latency for booting 200 CVMs concurrently for new functions is 28.5s, failing to meet the quality of service during load spikes. Note that the concurrent booting problem does not exist on TDX platforms, but launching a single TDX CVM is also slow (1.8s).

Existing work [56, 113, 43, 36, 119] uses VM checkpoint/restore to quickly boot VMs for functions. Yet, latest SEV/TDX still has no such support. CVMs may add hardware/firmware support for checkpoint/restore, but it will not be fast due to memory measurement and re-encryption (just like reconstructing a new CVM).

The problem of resource provisioning for warm start. For better performance, warm start techniques [39, 44, 59, 78] can be employed to avoid booting CVMs from scratch on the critical path. This typically involves maintaining a pool of pre-launched CVM instances [42]. Unfortunately, a resource-efficiency problem arises due to CVM's isolation properties: memory sharing between CVMs is strictly prohibited. Consequently, caching function instances with CVMs necessitates significantly higher resource allocation compared to traditional microVMs. As depicted in Figure 4a, caching 500 traditional microVMs might require only 12.5GB of memory, whereas caching 500 SEV microVMs could balloon 42.5GB of memory. This is because traditional VMs can share the guest OS text and read-only data to reduce the memory overhead [36, 93]. Besides, prior work has shown that sharing common code among function instances can yield memory savings of up to 50MB per instance [109]. The main reason why different CVMs cannot share memory is that each CVM uses a unique memory encryption key.

Moreover, one serverless application may contain a chain of functions [110, 125]. When functions run in different

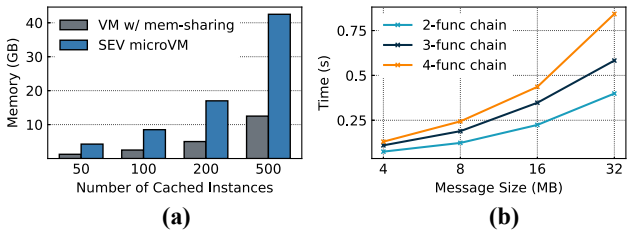

Figure 4. (a) Memory overhead of cached CVM instances, (c) inter-CVM communication cost.

CVMs, transferring data across functions involves repeated copies, encryption, and authentication, leading to additional overhead. Figure 4b shows that communicating a 32MB message across 4 CVMs takes 850ms.

Design-choice-2: Per-Tenant-CVM. To alleviate the conflicts between boot latency and resource provisioning, an alternative design is to run all functions of one tenant in one CVM. This allows memory sharing and function forking within each CVM, and supports inter-function interaction with shared memory within each CVM. However, it has two major disadvantages. First, the isolation between functions in the CVM is weak because a compromised function could exploit vulnerabilities in the guest Linux (due to its large code base). Although the functions in the same CVM belong to the same tenant, they usually serve different mutually distrusted end-user requests. Second, this design also leads to double cgroup problem. The host Linux sets the cgroup for the pre-warmed per-tenant CVMs and the guest Linux in the CVM sets the cgroup for function instances. Since the number of function instances in each CVM is dynamic (auto-scaling), it is difficult for the host to set a static cgroup for each CVM, and additional efforts are required to dynamically adjust the resource limitation of each CVM.

Design-choice-3: Per-Function-CVM. A third design is to run all instances of the same function in one CVM, While this appears to be a middle ground between the previously discussed designs, it inherits limitations from both. In such a design, both security isolation and resource efficiency are not optimal (weak inter-instance isolation and duplicated guest kernel memory overhead).

2.3 OS Requirements for Serverless Functions

To characterize the OS functionalities required for serverless computing, we analyze a real-world dataset of functions. We collected representative functions (over 150 in total) written in dominant languages (Python or Node.js) from three sources: (1) deployed functions from the industry [34, 8] (i.e., AWS Lambda, Microsoft Azure, and Google Cloud Functions); (2) open-source functions from the OpenFaaS community [28] (with 24.5K GitHub stars); and (3) benchmarks from academia, including ServerlessBench [125], FunctionBench [81], SeBS [55], and FaaSdom [99].

Analysis. We examine the dataset of collected functions in four aspects. (1) *Process management:* The auto-scaling feature allows functions to achieve high concurrency by automatically provisioning multiple instances, instead of requiring explicit multi-threading or multi-processing within the code. This eliminates the need for complex mechanisms like multi-user support, multi-process handling, and POSIX inter-process communication. (2) *Storage:* Functions are stateless, so they do not persist data in local storage. Instead, they access external storage like S3. Their only local storage dependency is the container filesystem (rootfs), which holds the function code and libraries. Additionally, temporary files can be stored in a temporary filesystem (tmpfs). In theory, functions only require rootfs (read-only) and tmpfs for filesystem access. (3) *Network:* Functions are event-driven. Thus, the minimal OS requirement is network access via sockets to allow functions to access external components of the serverless platform, such as object storage. (4) *Data-plane services:* There is a clear separation between control-plane and data-plane in serverless computing. The serverless platforms are responsible for the control-plane operations like load balancing, fault tolerance, function-chain orchestration, which lean on OS services such as resource monitoring, container managements, device managements, etc. Thus, these OS services can be removed from the data-plane OS within CVMs. In summary, the simplicity of the serverless programming model means functions typically depend on only a few OS functionalities.

Empirical validation. To validate our analysis, we use `strace` to trace system calls from the function dataset. As shown in Table 1, only 74 out of the 362 syscalls (in Linux 6.2) are used by serverless functions. Most syscalls are used to access the container's rootfs or network. Among the 74 syscalls, there are 15 syscalls that can be handled with a dummy implementation. These syscalls are used by language runtimes to support general (non-serverless) use cases. For example, the runtime may use `getpid` for logging and debugging and `fcntl` to close file descriptors after `execve`. The actual functionality of these syscalls is not needed by serverless functions.

Inspired by AWS Firecracker [38], a simplified hypervisor dedicated to serverless, we believe a simplified serverless function-oriented OS for CVMs is also desirable. For example, OS functionalities such as filesystems and network stacks can be excluded by securely reusing the host OS I/O stack with appropriate encryption and authentication.

This study of syscalls suggests exploring alternative guest OS designs for CVMs, instead of relying on a full-fledged Linux distribution. A tailored monolithic OS might be possible, but addressing complex kernel component dependencies remains a challenge [84]. To achieve an even smaller system-level TCB, we propose to use an exokernel-like [57] architecture (i.e., microkernel + libOS), leading to the design of CoFunc.

Table 1: System calls used by serverless functions.

Syscall	Number	Syscall	Number
FS	26	Event	4
Network	12	Misc	5
Memory	6	Dummy	15
Sync	3		
Thread	3	**Total**	**74**

3 Overview

3.1 The Architecture of Split Container

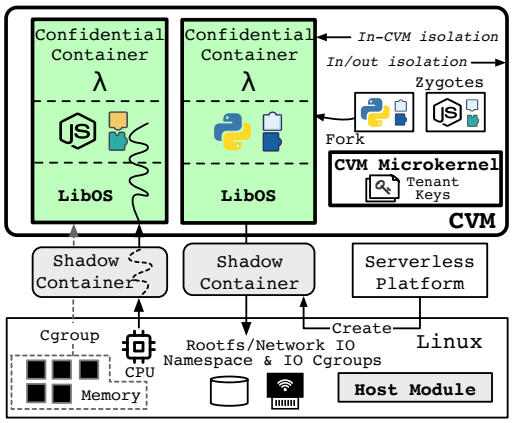

Figure 5. The architecture overview of split container.

Design implications. We first summarize the design implications of constructing confidential containers with CVM for serverless functions, according to our design space exploration. First, to minimize the TCB, a full-fledged OS like Linux should be kept outside the TCB of confidential containers. Second, to support a scalable number of function instances, various functions should be accommodated in one CVM, avoiding the hardware limitation and enabling memory sharing. Third, to minimize the function startup latency, CVM boot procedure and intensive attestation must be excluded from function launching. Fourth, to preserve functionality without TCB bloating for functions, most system calls can be redirected to the untrusted Linux while the others are supported by a small trusted kernel with a function-oriented libOS.

Basic Idea. We propose CoFunc, a CVM-based confidential container system for serverless functions. CoFunc runs different function instances (confidential containers) within one CVM for instant function boot and resource efficiency, and minimizes the TCB for inter-container isolation (the CVM kernel) with a novel *split container* design as follows.

(1) *Split container management (outside the CVM) and container protection (inside the CVM).* To maintain compliance with the existing container ecosystem, container management functionalities, including CPU/memory cgroups and isolated rootfs/networking, should be supported for confidential containers inside the CVM. Instead of implementing these functionalities within the CVM, CoFunc transparently passes through containerization offered by the host into the CVM, which is different from existing container architectures.

(2) *Offer all the required syscall functionalities with a minimized TCB by using I/O delegation and an exokernel-like architecture.* By identifying the small set of system calls required by the serverless function execution, CoFunc offers them while minimizes the TCB in a two-fold way. First, IO syscalls are delegated by the host OS instead of implemented inside the CVM. Second, CoFunc adopts an exokernel-like architecture (microkernel + libOS) in the CVM to offer the rest of the syscalls, which can keep the CVM OS kernel minimized.

System architecture. As depicted in Figure 5, each confidential container inside the CVM is paired with a *shadow container* outside the CVM. The host OS and serverless platform are still responsible for constructing (shadow) containers and allocating resources to them. Meanwhile, the real functions run inside confidential containers, which are hidden from Linux because of the CVM protection. A microkernel operates in the kernel mode of the CVM, establishes different address spaces for different confidential containers, and ensures the isolation between them.

A sharing-CVM design for instant function boot and resource efficiency. CoFunc runs different function instances within one CVM and has the following advantages. First, function instances are launched in the existing CVM, which avoids the performance bottleneck of CVM (concurrent) cold boot. Second, the different function instances running in the same CVM can share confidential memory, which enables CoFunc to boot multiple function instances by copy-on-write forking a single template instance, called *Zygote*, avoiding the memory consumption of keeping per-instance warm CVMs. §4.4 describes the techniques of *Zygote-based boot* and *split-attestation* for reducing the initialization and attestation overhead of function startup. Third, CoFunc supports chained functions to transfer intermediate data with shared memory instead of an untrusted external object storage, eliminating networking and data encryption overhead. The sender and receiver authenticate each other using a shared secret provided by the tenant, and then establish a trusted channel with the help of the CVM microkernel.

A split container design for strengthening inter-container isolation. CoFunc decouples container management from security protection. Specifically, Linux is still responsible for CPU scheduling and memory allocation for shadow containers, while a confidential container will inherit both CPU and memory resources from its corresponding shadow container (§4.1 and §4.2). This *resource granting* technique avoids re-implementing Linux container resource management mechanisms in the CVM microkernel, which reduces the TCB size. It also avoids the double cgroup problem (§2.2), because the host Linux only needs to set a cgroup for each shadow con-

tainer on demand without setting a cgroup for the CVM.

To avoid introducing complex I/O stacks inside the CVM, CoFunc uses *I/O delegation*: the shadow container delegates the I/O operations on Linux under the confinement of namespaces and I/O cgroups, and then returns the results back to the confidential container; the libOS employs end-to-end encryption and authentication for the I/O protection. CoFunc adopts an exokernel-like architecture in the CVM instead of using a full-fledged OS. Within the CVM, the CVM microkernel is the only TCB for intra-CVM isolation and a user-mode libOS serves the rest syscalls issued by the function (§4.3).

3.2 System Workflow

Key dispatching. Encryption keys are used to protect I/O, including network communication and container images authentication. CoFunc leverages remote attestation to securely distribute tenant keys to different CVMs. Remote attestation is offered by the CVM hardware for tenants to ensure the CVM is faithfully launched and establish secure channel with it [108, 105]. Figure 6a illustrates how to dispatch a tenant's key.

For simplicity, a serverless provider uses a coordinator server node for function uploading and scheduling as well as a group of invoker nodes that execute functions. CoFunc requires the coordinator node to initiate a key management service that manages tenant keys. The key management service runs inside a CVM. During registration, a tenant remotely attests the key management service (①), establishes a secure channel with it, and then sends his/her key to it (②). After the coordinator node instructs an invoker node to launch the CVM (③), the key management service attests the CVM to ensure that the CVM microkernel is faithfully booted (④), and then securely sends the tenant key to the CVM (⑤).

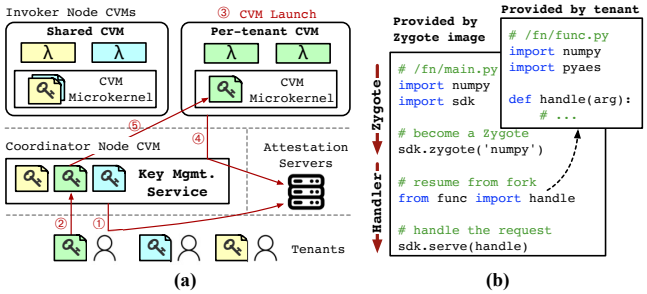

Figure 6. (a) Key dispatching, (b) A Zygote example.

Container image building. Tenants build container images of function handlers based on Zygote images. Typically, a Zygote image includes libraries for communicating with the serverless platform, and a language runtime with common libraries, such as Python and NumPy. A Zygote image also includes the binary of the program that runs in the shadow container. Tenants further install function code (i.e., a request handler) and other necessary libraries into the container images. CoFunc provides a toolchain to sign the container and Zygote images with the tenant key (§4.4), and the integrity of the images will be verified in the CVM during container launching. Tenants can also use the toolchain to encrypt selected sensitive files in the container images.

Container lifecycle. To launch a confidential container, an invoker node creates the shadow container in a similar way as normal containers are launched. The shadow container then communicates with the CVM microkernel to instantiate the confidential container. CoFunc implements the communication through a host kernel module (host module for short).

The CVM microkernel provides two interfaces for creating confidential containers: *fork* and *launch*. Zygote images can be launched inside the CVM in advance, and then become Zygotes for forking confidential containers (Figure 6b). The CVM microkernel also supports launching a confidential container without Zygotes. It maps the libOS inside the confidential container to load the code from the rootfs of the container image outside the CVM. Both interfaces will check the integrity of the loaded code (§4.4).

Once a shadow container is scheduled by Linux, it transfers the CPU control flow to its confidential container with the assistance of the host module. During execution, the confidential container may inform its shadow container to perform I/O operations through the host module. After the confidential container finishes execution, the CVM microkernel tears down its states, and transfers the CPU control flow back to the shadow container, which then exits.

3.3 Threat Model

In our threat model, an attacker may (1) compromise host-side privileged software, including Linux/KVM and serverless platforms, (2) conduct physical attacks to steal sensitive data [124, 67], or (3) deploy a malicious confidential container, attempting to attack other containers by compromising the CVM microkernel or through side channels [126, 83, 97]. The attacker aims to inspect (compromising confidentiality) or manipulate (compromising integrity) the private memory and register states of serverless functions. CVM implementation bugs [91, 89] and denial-of-service (DoS) attacks are beyond the scope of this paper.

Component trust relationship. A function trusts the CVM microkernel and all the code within its confidential container, including libraries (forked from Zygote) and the libOS. It by default does not trust other confidential containers or any shadow container. The CVM microkernel does not trust confidential containers or the software outside the CVM. It ensures the isolation among confidential containers. Function tenants trust the CVM and the key management service. The host OS does not trust the shadow containers or any software running inside the CVM.

4 Detailed Design

4.1 CPU Resource Granting

Linux regulates container CPU usage via CPU cgroups. Rather than duplicating such cgroups inside the CVM, CoFunc relies on Linux to schedule all the confidential containers in the CVM. It establishes one-to-one mappings between the threads in the shadow container (shadow threads) and the threads in the confidential container (confidential threads). This is achieved by letting a confidential thread exclusively occupy one CVM virtual CPU (vCPU) corresponding to its shadow thread. As illustrated in Figure 7, a shadow thread uses the VMEnter interface of the host module to activate a vCPU (i.e., become the vCPU) to run the corresponding confidential thread. The confidential thread runs with the CPU timeslices allocated to the shadow thread, so that the amount of CPU resources is controlled by the cgroup of the shadow container.

vCPU allocation. A shadow thread cannot become a hotplugged vCPU to enter the CVM because the CVM firmware disallows such an operation that may compromise its execution flow [3]. Therefore, CoFunc prepares a vCPU pool upon booting the CVM. Until a shadow thread activates it, a vCPU in the pool remains inactive and does not waste CPU resources. The pool requires only a minimal amount of memory to record vCPU contexts. For instance, a pool of 1,000 vCPUs would occupy only 11.7MB memory.

vCPU execution. At the CVM boot time, all vCPU contexts are initialized to wait for a confidential container creation request. Meanwhile, the host module marks each vCPU as idle. When a shadow container is created, the shadow thread needs to activate one idle vCPU through the host module to create the confidential container. Once the host module allocates an idle vCPU to a shadow thread, it marks the vCPU as busy. The activated vCPU launches a confidential container and turns into the confidential thread to execute the function.

Whenever the vCPU is interrupted by the timer, Linux will see the interrupt occurs in the host module where the shadow thread enters the CVM, and perform a reschedule if the current shadow thread has exhausted its timeslice. Upon being scheduled again, the shadow thread resumes its corresponding vCPU, and the interrupted confidential thread continues execution. In this way, the confidential thread inherits the scheduling context of the shadow thread, including its timeslices and priority.

After the confidential thread completes execution, it traps into the CVM microkernel, and the corresponding vCPU returns to a state of waiting for the next confidential container creation request. The vCPU returns control back to the shadow thread, and the host module marks the vCPU as idle once again.

We implement CPU resource granting based on KVM vCPU scheduling [50], which minimizes the need for modifications in Linux.

Security. Both Linux and the shadow containers cannot directly control the execution context of CVM vCPUs due to CVM protection. They can only activate or resume vCPU execution. Thereby, they cannot compromise the confidentiality or integrity of the confidential threads' execution context.

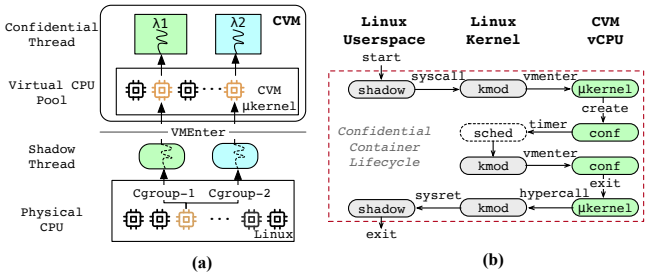

Figure 7. (a) CPU resource granting mechanism, (b) the CPU control flow of a confidential container (I/O delegation is not included for simplicity), shadow: shadow container, conf: confidential container, kmod: host module, μkernel: CVM microkernel.

Multi-threading. Although single-threaded function is a common practice because concurrency can be readily achieved through function auto-scaling [125, 60, 80], a function may use a multi-threaded language runtime (e.g., Node.js). Specifically, a Node.js function comprises a main thread and several passive threads that are transparently created by the runtime for processing I/O requests. By using thread synchronization primitives, the main thread can awaken passive threads, dispatch requests, and await their completion.

In such cases, if the main thread and passive threads are mapped to different vCPUs (one-to-one mapping), thread switching would necessitate vCPU switching, which results in increased overhead due to VM exits (e.g., a switch between TDX CVM and host takes about 5µs on Intel Sapphire Rapids CPU). Therefore, for Node.js functions, CoFunc runs both the main thread and passive threads of a confidential container on a single vCPU (corresponding to one shadow thread). The libOS offers the needed thread synchronization primitives and switches threads in a cooperative way without VM exits.

4.2 Memory Resource Granting

Linux regulates container memory usage via memory cgroups. A memory cgroup tracks all physical memory pages it owns and can be configured with a hard limit to specify the maximum allowable memory usage. CoFunc reuses the memory cgroup mechanism in Linux to control the memory usage of confidential containers. It does this by transforming the memory allocation of a confidential container into that of the corresponding shadow container, which is allocated with a memory cgroup. Thus, the memory usage of the confidential container is accounted for within the same cgroup.

This is accomplished by adapting the VM memory al-

location mechanism in KVM. KVM establishes bindings between a VM's guest physical address (gPA) range and its host daemon's (QEMU process) virtual address (hVA) range. Similarly, CoFunc establishes bindings between different gPA ranges of the CVM and the hVA ranges of different shadow containers, as illustrated in Figure 8a. The two ranges share the same host physical address (hPA) mapping. Specifically, when launching a confidential container, the CVM microkernel allocates a gPA range as its memory pool, the size of which is determined by a boot parameter set by the shadow container. The shadow container sets the hard limit of its memory cgroup as the size parameter. The CVM microkernel requests the host module to establish a binding between the gPA range and a hVA range of the shadow container.

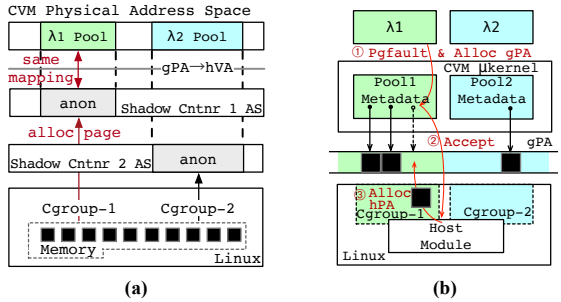

Figure 8. (a) Memory resource granting mechanism, (b) the workflow for handling a page fault of a confidential container, AS: address space, Cntnr: container, anon: anonymous mapping, pgfault: page fault, μkernel: microkernel.

During execution, a confidential container may trigger page faults because the CVM microkernel employs on-demand paging. As illustrated in Figure 8b, to handle a page fault for a confidential container, The CVM microkernel allocates memory (gPA) from the confidential container's memory pool and maps the gPA in the confidential container's page table. The CVM microkernel also maintains the metadata of each memory pool to keep track of whether a given gPA in the pool is mapped to a physical page (hPA). If the allocated gPA is not already mapped to a physical page, The CVM microkernel uses the CVM primitive (e.g., SEV *pvalidate* instruction) to add the mapping. The primitive (1) first triggers a nested page fault that instructs the host module to allocate a physical page, map it to the gPA in the CVM's nested page table (NPT), and set the CVM as the owner of the page (e.g., SEV *rmpupdate* instruction), (2) then accepts the physical page into the CVM. The host module handles the nested page fault as if it occurred within the hVA range of the corresponding shadow container, thereby charging the newly allocated physical page to the shadow container. To reduce VM exits due to frequent nested page faults, the CVM microkernel uses the primitive to add a huge page mapping in the NPT each time, and flags the relevant gPAs as mapped. If the physical page allocation fails (e.g., the hard limit of the cgroup is exceeded), the CVM primitive returns an error code and the CVM microkernel terminates the confidential container.

When a confidential container completes its execution, the memory resources it has consumed are reclaimed. The memory pool is recycled, and all the allocated physical pages are relinquished back to Linux when the shadow container exits.

Security invariants. Although Linux is responsible for filling the NPT of a CVM, both memory aliasing and remapping attacks are prevented in CoFunc. Memory aliasing is precluded for two invariants: 1) The CVM hardware ensures that a single physical page cannot be mapped to multiple gPAs in an NPT; 2) The CVM microkernel guarantees that the memory of mutually-distrusted confidential containers do not overlap. Remapping attacks are averted because of the third invariant: 3) The CVM microkernel must accept the NPT mappings prior to use, and the CVM microkernel never accepts any mapping that targets an already mapped gPA.

Table 2: System calls supported by CoFunc.

Type	System calls
FS	open/close, read/write, stat, mkdir, readlink, ...
Network	socket, bind, connect, sendto/recvfrom, ...
Memory	mmap*, munmap, mremap, mprotect, brk, madvise*
Thread	clone, yield, gettid
Event	epoll_(create,ctl,pwait), poll
Sync	pipe2, eventfd2, futex*
Misc	clock_get(time,res)*, getrandom, arch_prctl*, uname

* Partial support: some variants are not implemented.

4.3 Implemented Syscalls in Function-Oriented OS

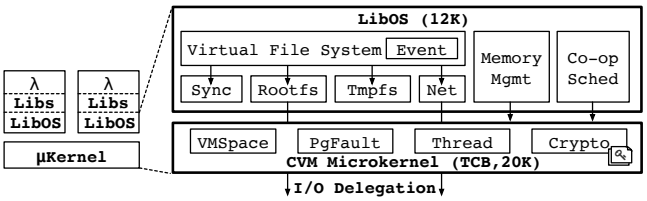

Figure 9. System calls implementation, VMSpace: virtual memory space, PgFault: page fault, Co-op: cooperative.

The libOS provides 74 syscalls according to the study presented in §2.3. Table 2 lists them (excluding the dummy ones) and 6 syscalls are partially supported because certain functionality, like writable `mmap` of rootfs files, or fork-related `madvise`, is not needed by functions.

Memory management. The libOS offers syscalls on updating the virtual memory space (such as `mmap/munmap/mprotect/brk`) to the confidential container. It transparently allocates or frees virtual memory regions of a confidential container, and requires the CVM microkernel to update the confidential container's page table. Given that serverless functions are ephemeral, The CVM microkernel does not incorporate advanced features like memory migration or compression. Catering to the

needs of functions, it implements the page fault handler for on-demand paging and copy-on-write for fork-based startup.

Threading and synchronization. All confidential threads of a confidential container run on a single vCPU (§4.1). The libOS implements the synchronization primitives and a cooperative scheduler. On synchronization events, the user-mode scheduler asks the CVM microkernel to switch confidential threads running on the vCPU. We choose to do the thread switching in kernel mode because we can use lazy floating-point unit saving.

Virtual file system (VFS). The libOS implements a virtual file system to dispatch syscalls based on file descriptors to different subsystems, including file system, network, and synchronization (`pipe/eventfd`). The VFS contains a table that records the metadata of file descriptors, such as descriptor type and cursor position. The VFS also supports event polling (`epoll/poll`) on network and synchronization file descriptors.

Network. The libOS handles network syscalls by delegating the network I/O operations to the shadow container. To protect network data, the libOS supports to transparently encrypt/decrypt the data with tenant keys.

File system. Serverless functions use the rootfs to load code and read-only data, and the temporary filesystem (tmpfs) to save temporary data. The libOS supports rootfs reading by delegating the file I/O operations to the shadow container, and the attestation mechanism (§4.4) ensures file integrity. The tmpfs is implemented inside the libOS to eliminate VM exits and data encryption.

Miscellany. `clock_gettime` is implemented with `rdtsc` instruction, and the host module provides the initial real-world timestamp on confidential container creation. `getrandom` is implemented with `rdrand` instruction. `arch_prctl(SET_FS)` is implemented based on `wrfsbase` instruction.

I/O delegation. For one delegation (the shadow container helps the confidential container to load files from the container rootfs or send/receive network packets), The libOS first prepares the syscall parameters in a shared buffer between the confidential thread and the shadow thread. It then issues a hypercall to transfer CPU control flow to the host module, which requires the shadow thread to take over. The shadow thread reads the shared buffer and invokes the host syscall to complete the delegation request, sets the results in the buffer, and resumes the confidential thread via the host module.

CoFunc also supports polling-based I/O delegation [114, 13] to avoid the VM exit overhead. This is achieved by creating a thread in the shadow container that polls the shared buffer and handles syscalls, which, however, leads to more CPU utilization.

4.4 Fast Function Boot with Split Attestation

Zygote-based boot. CoFunc optimizes the startup of the confidential container to meet the fast boot requirement of functions. First, it eliminates the CVM booting overhead from function booting by using the pre-launched CVM to run functions. Second, it accelerates shadow containers startup on Linux with lean container techniques [103], which tailor container construction for functions. Third, it permits forking confidential containers from Zygotes in the CVM to bypass most of the code loading and measurement. Specifically, it creates Zygotes that include the language runtime and libraries in the CVM in advance. A shadow container can invoke the *fork* interface of the CVM microkernel with a designated Zygote, and the CVM microkernel will create the confidential container by forking the Zygote in a copy-on-write way (reduce latency and save memory). After that, the Zygote child (confidential container) loads and executes the function code.

Because Node.js is a multi-threaded runtime, the *fork* interface also supports multi-threaded Node.js Zygotes. This is implemented by cloning all the threads besides marking the address space as copy-on-write.

Split-attestation. By using Zygote-based forking, the attestation process for a function is split into three phases. First, a CVM for confidential containers undergoes attestation, ensuring that the CVM is running with a trusted CVM microkernel. Second, CoFunc prepares and verifies Zygotes. These two phases are performed out of the critical path of running a function in a confidential container. Finally, when booting a function, CoFunc only needs to load and verify the function code.

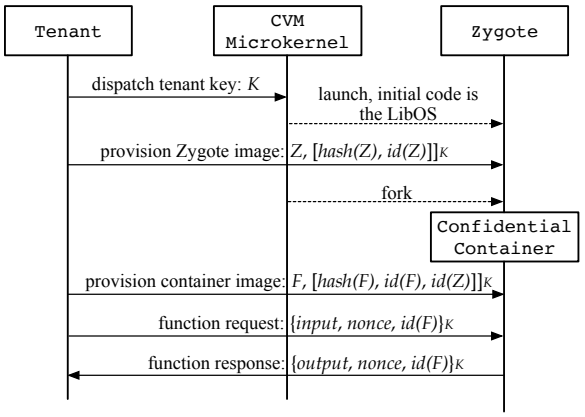

Figure 10. Attestation workflow of a confidential container (fork mode). K: tenant key, F: container image, Z: Zygote image, $hash()$: hash value of each file in the image, $id()$: image ID, $\{\}_K$: encrypted with K, $[]_K$: signed with K.

Signature. Each container/Zygote image contains a metadata file signed with the tenant key. The metadata file contains the hash value of each file in the image and a unique ID of the image. The metadata file of a container image also

contains the ID of its base Zygote image. The libOS checks the hash value when loading files from the images, but it cannot access the tenant key to verify the signature of the metadata file, otherwise a vulnerable confidential container could leak the key. The tenant key is stored in the CVM microkernel, so that the libOS needs to invoke the CVM microkernel to verify the signature.

Confidential container attestation. Figure 10 illustrates the attestation workflow of a confidential container.

After a CVM is launched, the tenant attests the CVM microkernel and dispatches the tenant key to it, with the help of the KMS (§3.2). The tenant uploads his/her Zygote images, container images, and the metadata files to the serverless platform.

The CVM microkernel launches a Zygote with the libOS as the initial code. The libOS code is embedded in the CVM microkernel so that it has already been attested by the tenant. The Zygote loads and verifies the metadata file from its image. It then loads the language runtime and libraries and executes the initialization code. When a file (e.g., a library) is loaded, its content is verified against the hash value in the metadata file.

When the tenant invokes a function, he/she encrypts the request with the tenant key before sending it to the serverless platform. The request consists of the input data, the ID of the container image, and a randomly generated nonce.

The CVM microkernel forks a Zygote belonging to the tenant to create a confidential container. The confidential container loads and verifies the metadata file from its image. It then checks that its parent Zygote matches the Zygote image ID in the metadata file. After that, it loads and verifies the function code from its image. It decrypts the invocation request and checks that its image ID matches the one in the request. Finally, it handles the invocation and encrypts the response with the tenant key.

The response contains the function output, the nonce value, and the ID of the container image. The tenant decrypts the response and verifies the nonce and the ID, which can prevent replay attacks.

Caching. According to existing studies [103, 80], the popularity of libraries used by serverless functions and the invocation frequency of various functions show great skewness, which can help to decide which Zygotes to cache in advance. The Zygote caching policy is orthogonal to this paper. If a function cannot be forked from some cached Zygote, the CVM microkernel also offers the *launch* interface to boot a confidential container from the scratch.

5 Implementation

CVM hardware. We have implemented a prototype of CoFunc on both AMD SEV and Intel TDX machines. It requires the CVM hardware to offer secure memory, secure vCPU context, VM enter/exit, secure nested-page-table mapping, CVM-host memory-sharing, and remote attestation. These features are also provisioned by other CVMs like ARM CCA. The confidential container with the libOS and the CVM microkernel operate in user and kernel mode, respectively, inside the CVM (e.g., TDX SEAM non-root mode). The shadow container and Linux operate outside the CVM (e.g., TDX non-SEAM root mode). CoFunc does not need to modify CVM firmwares like TDX SEAM module or SEV PSP firmware.

Implementation effort. The CVM microkernel is built over ChCore microkernel [66, 7, 122] and contains about 20K LOC. We add CVM support and function-specific customization to the microkernel. We also add a crypto library to the microkernel, which supports AES-128 and SHA-256 and contains about 2K LOC. Additionally, we extend the userspace musl libc [26] with the libOS, adding around 12K LOC for syscalls implementation.

Usability. If functions require additional syscalls not covered by CoFunc, the libOS can be extended to include new syscalls. CoFunc currently focuses on CPU TEEs and we leave the support of heterogeneous TEEs for GPU workloads like ServerlessLLM [61] as future work.

6 Security Analysis

We argue that CoFunc achieves comparable security guarantee to Kata-CVM, i.e., a per-instance-CVM design (the strongest isolation).

6.1 Similarity: CVM Protection

As confidential containers run in CVMs no matter in CoFunc or Kata-CVM, they are always isolated from software outside the CVM including the host Linux. Thereby, both designs are robust to Linux CVEs, which are usually exploited by malicious containers for privilege escalation in the non-confidential container scenarios. Meanwhile, owing to the usage of CVM, the memory of each confidential container is encrypted, and only gets decrypted inside the CPU, which prevents physical attacks such as cold-boot [67] and bus-snooping [124]. The CVM also protects the hardware primitives within it, such as a monotonous timer and CPUID [20, 2].

6.2 Difference-1: CVM Sharing

Mutually-distrusted functions run in the same CVM in CoFunc, while they are separated into different CVMs in Kata-CVM. This leads to several differences.

Software TCB. CoFunc uses both hardware-based and software-based security isolation, i.e., CVM hardware prevents threats from outside while the CVM microkernel guarantees the isolation among different functions. The CVM microkernel is an extra software TCB introduced by CoFunc. Our practice indicates that the kernel's responsibilities are clear, requiring only 20K LOC, which could be a feasible size for formal verification [82]. If new syscall implementations are required, they can be incorporated into the libOS

while keeping the CVM microkernel small.

Memory encryption key. CoFunc makes various functions share the same memory encryption key while Kata-CVM does not, which may lead to more severe consequences of successful physical attacks, e.g., one key leakage will endanger all the functions. Yet, Intel has released MKTME [19], which allows to configure various memory encryption keys within one CVM and could be a natural fit for CoFunc. To use multiple keys in one CVM, additional mechanisms need to be incorporated into the open-sourced TDX module (future work).

Side-channel attack. CoFunc may facilitate cache-timing attacks [126, 98] by allowing confidential containers within a CVM to share memory for common libraries. To mitigate this risk, the default security policy in CoFunc prohibits cross-tenant memory sharing (i.e., different tenants operate with different Zygotes). Orthogonal techniques like side-channel-free libraries [48] could also be employed.

Transient execution attacks, like Meltdown [97], Spectre [83], or L1TF [51], can be exploited to breach isolation domains. Different from Kata-CVM, CoFunc must prevent transient execution attacks within one CVM. Meltdown and L1TF have been fixed with hardware or microcode revision [53], while Spectre can be mitigated through standard methods [31]. Specifically, the CVM microkernel executes the IBRS instruction before processing a syscall to prevent container-to-kernel attacks, and the IBPB instruction before launching a confidential container to prevent cross-container attacks.

Furthermore, as CoFunc runs multiple containers on a shared kernel, a malicious container may attempt to create a side channel by monitoring the resource statistics (e.g., memory usage) of other containers [77, 116] through the kernel interfaces. To eliminate this risk, the CVM microkernel does not provide resource monitoring interfaces (e.g., procfs), as they are unnecessary for serverless functions.

6.3 Difference-2: Split Container

CoFunc relies on the host kernel to manage the CPU and memory resource of confidential containers, and handle their I/O requests. This requires several security considerations.

Resource granting. CPU granting can only cause DoS as execution contexts are managed inside CVM (§4.1). Aliasing or remapping attacks in memory granting can be prevented with standard CVM mechanisms like SEV RMP and TDX PAMT (§4.2).

I/O delegation. For file accessing, the confidential container only delegate syscalls for loading code from rootfs. The integrity of code data is verified upon loading (§4.4). For networking, end-to-end data encryption is employed to protect data confidentiality and integrity. Packet size exposure can be mitigated with data padding techniques [74].

Delegating syscalls to the host kernel may lead to Iago attacks due to malicious syscall return values. To mitigate this risk, CoFunc only delegates I/O syscalls, while handling syscalls prone to Iago attacks (e.g., mmap) within the CVM (§4.3). Additionally, the libOS verifies the return values of I/O syscalls before using them. For instance, it checks that the size of the received data does not exceed the buffer size.

7 Performance Evaluation

Testbed. Our evaluations are carried out on both Intel TDX and AMD SEV platforms. We run TDX experiments on an Intel server with 4 CPUs (Sapphire Rapids) at 4GHz with 48 cores, 503GB DRAM. The OS is RHEL 8.7 with Linux kernel 5.19.0. We run SEV experiments on an AMD server with 2 CPUs (EPYC 7T83) at 2.45GHz with 64 cores, 502GB DRAM. The OS is Ubuntu 20.04 with Linux kernel 6.1.0.

Baseline. 1) Vanilla Kata-CVM [22] on TDX and Kata-CVM optimized with specialized SEV microVM [70]. 2) Lean container [103, 121] (marked as Native), a non-confidential native Linux container optimized for serverless.

Benchmarks. For performance assessment, we utilize all 28 functions from four serverless computing benchmarks: ServerlessBench [125], FunctionBench [81], SeBS [55], and PIE-benchmark [90]. The functions are written in Python and Node.js, which are the most popular programming languages for serverless. All the function container images are built based on Alpine 3.17 image with musl libc 1.2.3.

7.1 End-to-End Latency

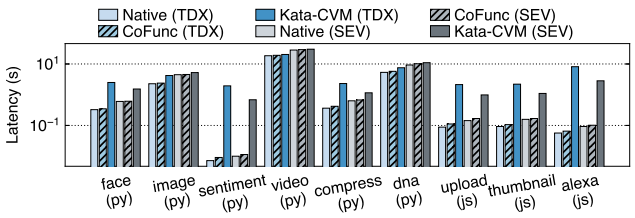

Figure 11. End-to-end latencies of serverless functions.

For all the 28 functions in the four serverless benchmarks, CoFunc outperforms the confidential baseline (Kata-CVM) significantly and has a moderate overhead compared to the non-confidential baseline (Native). Compared with vanilla Kata-CVM [22] on TDX and our optimized Kata-CVM with specialized SEV microVM, CoFunc shows an average 44× (1.06∼215×) and 12× (1.02∼60×) speedup in end-to-end latency. The reason for the large speedup span is that eliminating the CVM startup and attestation overhead benefits the shorter functions most and the function execution time differs from each other. Compared with the native container, CoFunc incurs 2.7%∼26.7% (TDX) and 0.7%∼17.3% (SEV) overhead, and the average overhead is 13.1% (TDX) and 7.8% (SEV). Due to space limit, Figure 11 shows the end-to-end latency for handling a single request for 9 representative functions like face detection and

sentiment analysis, which covers short-term/long-term functions, CPU-memory/IO intensive functions, and the best/-worst cases for CoFunc.

This end-to-end latency of a serverless function contains two parts, the startup stage and the execution stage. The startup stage can be further divided into: preparing the container environment (namespace/cgroup or virtual machine) (*Startup-Stage-1*), and loading and initializing the libraries and the functions (*Startup-Stage-2*). The execution stage includes both the handling of requests and the transferring of parameters and return values (local network). The performance advantage of CoFunc over Kata-CVM mainly comes from the startup stage, as analyzed in §7.2. Its overhead relative to Native resides in both stages and is analyzed in §7.2 and §7.3.

7.2 Breakdown: Startup Stage

7.2.1 Startup-Stage-1: Containerization

Speedup vs. Kata-CVM: The initialization of a Kata-CVM container takes 1.8s (TDX) and 334ms (SEV), which involves the complex CVM initialization and attestation. It results in a large overhead for short-term serverless functions. CoFunc avoids this overhead by sharing the pre-launched CVM among containers, reducing the containerization latency to <15ms. So, in startup-stage-1, CoFunc outperforms Kata-CVM by 120× (TDX) and 22.3× (SEV). Moreover, as is shown in Figure 12, CoFunc avoids the serial startup problem on SEV and outperforms Kata-CVM by 100× when 200 containers boot concurrently. The concurrent startup performance of CoFunc on TDX is similar to that on SEV.

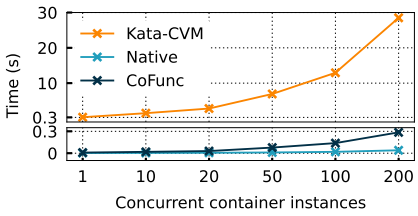

Figure 12. End-to-end latency of concurrent startup (SEV).

Overhead vs. Native: The initialization of a CoFunc container involves starting a shadow container (i.e., a native container with lean container optimizations) and a confidential container. The initialization of a shadow container takes 1.69ms (TDX) and 1.64ms (SEV), which is same as Native. For the confidential container, CoFunc needs to prepare the vCPU, memory pool, and shared memory, which incurs up to 10% (TDX) and 7% (SEV) overhead on all the 28 functions. This overhead is positively correlated with the memory pool size because a larger memory pool means initializing more pool metadata. For instance, it takes 0.71ms (TDX) and 0.90ms (SEV) to create a 16MB confidential container, and 10.2ms (TDX) and 8.3ms (SEV) to create a 1GB one.

7.2.2 Startup-Stage-2: Code Loading and Initialization

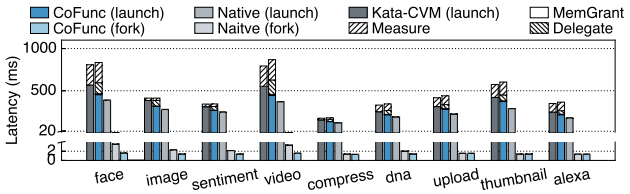

Figure 13. The startup-stage-2 latencies of functions (SEV) in launch and fork modes, Measure: code measurement, Delegate: I/O delegation, MemGrant: memory granting.

Figure 13 shows the startup-stage-2 latency of functions. Both CoFunc and Native support to *launch* a function from scratch or *fork* it from a Zygote.

Overhead vs. Native: In *launch* mode, the startup-stage-2 latency is high, especially when large libraries are imported. For example, the latency for CoFunc could be 657ms (TDX) and 895ms (SEV) on the face detection function which uses OpenCV library (200 MB). Besides loading files into memory and running library initialization logic, CoFunc incurs three extra overhead sources: (1) Delegate: I/O delegation (VM exit) overhead for rootfs access, (2) Measure: code measurement overhead (although using hardware accelerated SHA256), (3) MemGrant: memory granting overhead for allocating memory for libraries. Figure 13 gives the overhead breakdown.

In *fork* mode, the startup-stage-2 latency is reduced to <2ms for all the functions, because CoFunc only needs to fork the Zygote and load the function handler code. The function handler code loading and measurement is fast (<0.2ms) because the code size is small. Since the Zygote is pre-initialized and cached inside CVM, library loading/initialization and the corresponding delegation/measurement/memory granting overhead are all avoided. Fork mode is used for the evaluations in §7.1 and makes CoFunc show negligible overhead compared with Native in startup-stage-2. While CoFunc implements multi-threaded Node.js forking, it is not supported by Linux. For fairness, we use CoFunc forking latency for Native on Node.js functions.

Speedup vs. Kata-CVM: We also evaluate the startup-stage-2 latency of Kata-CVM. In startup-stage-2, CoFunc is 148∼499× (TDX) and 134∼513× (SEV) faster than Kata-CVM owing to Zygote-based booting and split attestation.

7.3 Breakdown: Execution Stage

7.3.1 Execution Overhead Compared with Native

Breakdown analysis. The execution stage overhead is divided into three parts: data encryption/decryption, memory granting and I/O delegation. Table 3 shows the breakdown. We abbreviate the function name to its initials.

Table 3: Execution stage overheads (TDX/SEV, %), Encrypt: data encryption/decryption, MemGrant: memory granting, Delegate: I/O delegation, #Exit: number of VM exits per ms.

Fn	Encrypt	MemGrant	Delegate	Others	#Exits
F	0.57/0.44	2.58/0.43	0.93/0.70	1.13/0.14	0.55
I	1.17/1.05	1.63/0.25	1.02/0.81	0.48/-1.55	0.74
S	0.00/0.00	15.6/3.36	2.62/2.71	-2.43/-1.83	2.40
V	0.19/0.20	0.40/0.08	0.19/0.13	1.95/2.73	0.09
C	6.54/6.02	2.77/0.50	2.99/2.28	2.11/-2.16	1.47
D	3.97/4.23	3.44/0.58	0.30/0.20	-0.43/1.47	0.12
U	10.0/11.2	12.6/2.24	2.87/1.61	-1.31/1.51	4.40
T	3.95/3.83	5.58/1.00	2.43/2.55	1.91/-0.99	0.98
A	0.00/0.00	7.94/1.46	2.15/2.25	0.76/2.10	1.68

Function parameters and return values undergo encryption before transmission to the untrusted serverless platform or external storage. The encryption and decryption within confidential containers leads to 2.9% (TDX) and 3.0% (SEV) overhead on average.

Memory granting happens due to on-demand paging and incurs memory allocating and accepting overhead. Especially, this affects functions processing large data (D, U) or functions involving frequent copy-on-write after forking (S, U, T, A). §7.3.2 shows the overhead reduction by using huge page granting.

The overhead of I/O delegation stems from VM exits and parameter copying. During function execution, network syscalls constitute the majority of delegated I/O. So, if a function does not involve intensive network syscalls, it will not severely suffer from the delegation overhead. I/O delegation poses <5% overhead for most of the functions. An exception in the 28 functions is *upload* that merely downloads a file from Internet and sends it to an object storage without any other logic. It is network intensive and suffers from 8.0% (TDX) and 8.7% (SEV) delegation overhead. To optimize it, we use the polling-based I/O delegation (§4.3) to eliminates the VM exit overhead, which reduce the delegation overhead on it to 2.9% (TDX) and 1.6% (SEV) (mainly caused by memory copying).

Other performance differences are attributed to implementation difference between the CVM microkernel and Linux, as shown in the following kernel microbenchmarks.

Table 4: Kernel microbenchmarks (iperf: GB/s, others: ns).

Testcase	Native		CoFunc	
	TDX	SEV	TDX	SEV
hypercall	/	/	4,900	2,700
rootfs read	223	434	8,211	5,982
iperf	70.4	45	29.5	22
page fault	883	950	857	1,078
tmpfs write	74	92	16	20
pipe	172	283	150	124
eventfd	98	156	20	33

Kernel microbenchmarks. Table 4 compares the performance of CoFunc and Native on kernel microbenchmarks. CoFunc performs comparably or better than Native for non-delegated operations such as tmpfs write, page fault handling (without NPT violation), and pipe read/write. For delegated I/O operations, including rootfs access and networking, CoFunc is notably slower due to the VM exit overhead which includes both hardware overhead (CVM exiting and re-entering) and software overhead (the host module). The hardware overhead of VM exit (a direct switch between CVM and host) is 4,900ns (TDX) and 2,700ns (SEV). The rest overhead is attributed to the software overhead, especially serving the VMEnter request from the shadow container. We currently reuse the existing KVM interface (`KVM_RUN`) for implementation, which entails various checks and could be further optimized in CoFunc.

7.3.2 Memory Granting Optimizations

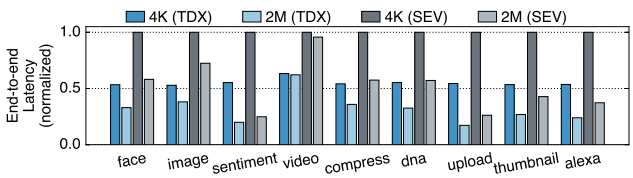

Figure 14. Effects of huge page granting optimization.

Granting memory from shadow containers to confidential containers needs the cooperation between the CVM microkernel and the host module. It happens when the CVM microkernel needs to allocate a gPA to a confidential container while the gPA has not mapped in NPT. During the procedure, the host module needs to allocate the granted page, and set the ownership of the granted page to the CVM with CVM primitives; the CVM microkernel needs to accept the granted page into the CVM with CVM primitives. Also, triggering the procedure involves a VM exit.

Granting a huge page (2M) instead of a 4K page each time helps to amortize the overhead, e.g., the page allocation/accept overhead and the VM exit overhead. Specifically, a single 2M granting takes 370µs (TDX) and 112µs (SEV), while a single 4K granting takes 15.7µs (TDX) and 31µs (SEV). Meanwhile, we also use persistent huge pages [18] to accelerate huge page allocation on Linux, reducing latency of 2M granting by 280µs (TDX) and 178µs (SEV). We also find that the memory accepting and page ownership setting operations on SEV are faster than those on TDX, which makes the memory granting overhead higher on TDX (§7.3.1).

Figure 14 shows the end-to-end latency of various functions under 4K/2M granting. Huge page granting reduces the latency by 42.2% (TDX) and 47.5% (SEV) on average.

7.3.3 Other Optimizations

In-CVM tmpfs. Figure 15 compares the end-to-end latency of functions with and without in-CVM tmpfs. Delegating I/O on tmpfs incurs VM exit and data encryption overhead. For functions that frequently access tmpfs (like *video processing* and *data compression*), using an in-CVM tmpfs reduces the total latency by 7.2% (TDX) and 10.6% (SEV) on average.

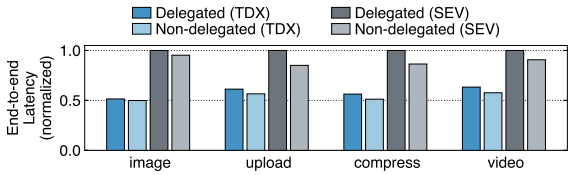

Figure 15. Effects of non-delegated tmpfs optimization.

In-CVM thread synchronization. As stated in §4.1, the libOS implements the thread synchronization syscalls for multi-threaded Node.js functions. An alternative design is to use Linux for thread pausing and awaking, while keeping security-related states (e.g., futex variables) in the CVM. To use this design, CoFunc needs to establish one-to-one mappings between enclave threads and vCPUs. The vCPU is paused when the enclave thread needs to wait for a synchronization event, and is awaken when the event arrives, which incurs VM exit overhead. By implementing thread synchronization in CVM, the end-to-end latencies of the 7 Node.js functions in the serverless benchmarks are reduced by 17.1% (TDX) and 11.1% (SEV) on average.

7.4 Function Chain and Memory Usage

Function chain optimization. We evaluate a data intensive chained application, FINRA [16]. It consists of two functions: a downloading function and an auditing function. The downloading function fetches large market data from a remote server and stores them to a Redis key-value (KV) store. Then 200 auditing functions is started concurrently to audit the data. In CoFunc, the data can be transferred by using shared memory instead of an external store. Compared with a no shared-memory version (CoFunc-KV), CoFunc is 4.6× faster since of the elimination of network and encryption overhead. It outperforms Kata-CVM with microVM optimization by 31× because Kata-CVM needs to boot new CVMs to run each function.

Empty containers memory usage. Kata-CVM (with microVM optimziation) incurs 85MB memory overhead per container when 100 empty containers are started, due to the guest Linux kernel, and the lack of memory sharing among different CVMs. In contrast, CoFunc only uses 4.7MB memory per container when 1,000 empty containers are deployed within one CVM. Compared with Native, CoFunc has a small memory overhead caused by the memory pool metadata (1MB for a 128MB container) and the shared memory buffer (128KB).

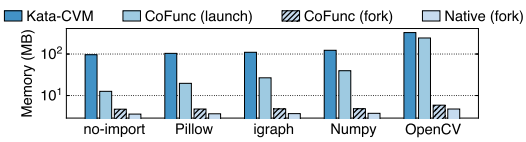

Figure 16. Memory usage of different Python containers.

Python containers memory usage. Figure 16 shows the average memory overhead of 200 Python containers. The memory usage of CoFunc in *fork* mode is less than that in *launch* mode because of memory sharing among confidential containers. Compared with Kata-CVM, CoFunc-*fork* achieves 20∼56× reduction in memory usage.

8 Other Related Work

Unlike confining containers within VMs like gVisor [12] and QuarkContainer [29], confidential containers aim to protect the confidentiality and integrity of containers even when both the privileged host OS or hypervisor is malicious. To this end, several prior work [40, 106, 63, 118, 49] uses SGX to protect serverless functions. Reuseable enclave [127] allows serverless platforms to rapidly reset and reuse an SGX enclave instead of relaunching it. TZ-Container [71] protects containers with ARM TrustZone while requires intrusive modifications on Linux, such as eliminating some privileged instructions or refactoring the direct mapping mechanism. CoFunc differs from them in leveraging CVM to build confidential containers for serverless computing.

HypSec [92] and TwinVisor [88] use ARM TrustZone to build hardware-software co-designed CVMs. Our design of CVM-based split container may also work with such CVMs. Komodo [58] isolates multiple SGX-compatible enclaves in ARM TrustZone with a formally-verified microkernel. CoFunc shares a similar design principle with them by combining hardware and software isolation.

Some prior work [52, 17] also runs serverless functions on libOSes for efficiency and small memory footprint. Yet, CoFunc leverages the architecture of microkernel with libOSes to minimized the TCB for inter-function isolation. Gramine-TDX [86] is concurrent work that designs a single-process tailored libOS for TDX CVMs, delegating I/O to the host kernel. It can be used as the libOS in CoFunc.

A prior study [120] reveals that some serverless platforms run multiple function instances of a single tenant in one VM. Our work enhances the security of such design and leverages it to address the mismatch between CVM and serverless.

Loupe [87] conducts a dynamic analysis on the OS features required by applications. It discovers that many applications can be supported by implementing a small number of system calls, which aligns with our findings on severless functions.

9 Conclusion

This paper analyzes the challenges arising from the mismatches between serverless computing and CVM hardware features, and presents a new container architecture called split container to address them.

10 Acknowledgment

We sincerely thank all the anonymous reviewers, whose reviews, feedbacks, and suggestions have significantly strengthened our work. This research was supported in part

by the National Natural Science Foundation of China (No. 62202292,62432010), the Fundamental Research Funds for the Central Universities, and research grants from Huawei Technologies and Intel. Corresponding author: Jinyu Gu (gujinyu@sjtu.edu.cn).

11 Ethical Considerations

We attest that the research team considered the ethics of this research. No participants other than the research team were involved in this research. The research did not involve activities that could negatively affect team members. No personal information was collected for this research. All evaluations were done on our local machines, not on live systems. The outcomes of this research are intended to improve cloud security, which are benign.

12 Compliance with the Open Science Policy

We openly share the research artifacts, including the source code of our system, the evaluation datasets, and the evaluation scripts. The artifacts are available at https://figshare.com/articles/software/CoFunc_Artifacts/28234346.

References

[1] AMD secure encrypted virtualization SEV. https://https://developer.amd.com/sev/.

[2] AMD64 Architecture Programmer's Manual. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf.

[3] Architecture specification: Intel trust domain extensions (intel tdx) module. https://cdrdv2-public.intel.com/733568/tdx-module-1.0-public-spec-344425005.pdf.

[4] Arm confidential compute architecture (arm cca). https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture.

[5] Aws step functions. https://aws.amazon.com/step-functions/.

[6] cgroups(7) — linux manual page. https://man7.org/linux/man-pages/man7/cgroups.7.html.

[7] Chcore microkernel github repo. https://github.com/SJTU-IPADS/OS-Course-Lab/tree/source.

[8] Cloud functions for firebase sample library. https://github.com/firebase/functions-samples.

[9] Confidential computing for everyone: Getting started with occlum and its related technologies. https://www.alibabacloud.com/blog/confidential-computing-for-everyone-getting-started-with-occlum-and-its-related-technologies_598671.

[10] Confidential containers. https://confidentialcontainers.org.

[11] Confidential containers with red hat openshift container platform and ibm® secure execution for linux. https://www.ibm.com/blog/confidential-containers-with-red-hat-openshift-container-platform-and-ibm-secure-execution-for-linux/.

[12] The container security platform — gvisor. https://gvisor.dev.

[13] Efficient io with io_uring. https://kernel.dk/io_uring.pdf.

[14] An epyc escape: Case-study of a kvm breakout. https://googleprojectzero.blogspot.com/2021/06/an-epyc-escape-case-study-of-kvm.html.

[15] Escaping virtualized containers. https://i.blackhat.com/USA-20/Thursday/us-20-Avrahami-Escaping-Virtualized-Containers.pdf.

[16] Finra adopts aws to perform 500 billion validation checks daily. https://aws.amazon.com/cn/solutions/case-studies/finra-data-validation/.

[17] Hardware accelerated applications on unikernels for serverless computing. https://archive.fosdem.org/2022/schedule/event/anano/attachments/slides/4924/export/events/attachments/anano/slides/4924/NBFC_microkernel_2022.pdf.

[18] Hugetlb pages. https://docs.kernel.org/admin-guide/mm/hugetlbpage.html.

[19] Intel architecture memory encryption technologies. https://cdrdv2-public.intel.com/679154/multi-key-total-memory-encryption-spec-1.4.pdf.

[20] Intel Trust Domain Extension Linux Guest Kernel Security Specification. https://intel.github.io/ccc-linux-guest-hardening-docs/security-spec.html.

[21] Intel trust domain extensions (intel tdx). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[22] Kata containers - open source container runtime software. https://katacontainers.io.

[23] Kata containers with AMD SEV-SNP VMs. https://github.com/kata-containers/kata-containers/blob/main/docs/how-to/how-to-run-kata-containers-with-SNP-VMs.md.

[24] Linux containers. https://linuxcontainers.org.

[25] Microsoft and red hat are collaborating in deploying confidential containers on the public cloud. https://www.redhat.com/en/blog/deploying-confidential-containers-public-cloud.

[26] Musl LibC. https://musl.libc.org/.

[27] namespaces(7) — linux manual page. https://man7.org/linux/man-pages/man7/namespaces.7.html.

[28] Openfaas: Sample functions. https://github.com/openfaas/faas/tree/e778a3a6ded22fef3a28e78e28d5dc33f0d17dfd/sample-functions.

[29] Quark: A secure container runtime with oci interface. https://github.com/QuarkContainer/Quark.

[30] runc. https://github.com/opencontainers/runc.

[31] rust Domain Security Guidance for Developers. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/trusted-domain-security-guidance-for-developers.html.

[32] Scontain. https://learn.microsoft.com/en-us/azure/confidential-computing/partner-pages/scone.

[33] Security guidelines for serverless computing. https://www.itu.int/ITU-T/workprog/wp_item.aspx?spm=a2c6h.12873639.article-detail.8.80136f63jL9AA3&isn=19108.

[34] Serverless examples. https://github.com/serverless/examples/tree/master.

[35] Sev secure nested paging firmware abi specification.

https://www.amd.com/content/dam/amd/en/documents/epyc-technical-docs/specifications/56860.pdf.

[36] What is vm templating and how to enable it. https://github.com/kata-containers/kata-containers/blob/main/docs/how-to/what-is-vm-templating-and-how-do-I-use-it.md.

[37] Attacks are forwarded: Breaking the isolation of MicroVM-based containers through operation forwarding. In *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA, Aug. 2023. USENIX Association.

[38] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, Feb. 2020. USENIX Association.

[39] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. SAND: towards high-performance serverless computing. In H. S. Gunawi and B. Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 923–935. USENIX Association, 2018.

[40] F. Alder, N. Asokan, A. Kurnikov, A. Paverd, and M. Steiner. S-faas: Trustworthy and accountable function-as-a-service using intel SGX. In R. Sion and C. Papamanthou, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop, CCSW@CCS 2019, London, UK, November 11, 2019*, pages 185–199. ACM, 2019.

[41] Alibaba cloud. Alibaba serverless application engine. https://www.aliyun.com/product/aliware/sae, 2022.

[42] Amazon Web Services. Configuring provisioned concurrency. https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html.

[43] L. Ao, G. Porter, and G. M. Voelker. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 730–746, New York, NY, USA, 2022. Association for Computing Machinery.

[44] Apache OpenWhisk Website. https://openwhisk.apache.org, 2022.

[45] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. Scone: Secure linux containers with intel sgx. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 689–703, USA, 2016. USENIX Association.

[46] AWS. Aws lambda. https://aws.amazon.com/lambda, 2022.

[47] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 164–177, New York, NY, USA, 2003. Association for Computing Machinery.

[48] G. Barthe, B. Grégoire, and V. Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 328–343, 2018.

[49] S. Brenner and R. Kapitza. Trust more, serverless. In M. Hershcovitch, A. Goel, and A. Morrison, editors, *Proceedings of the 12th ACM International Conference on Systems and Storage, SYSTOR 2019, Haifa, Israel, June 3-5, 2019*, pages 33–43. ACM, 2019.

[50] E. Bugnion, S. Devine, M. Rosenblum, J. Sugerman, and E. Y. Wang. Bringing virtualization to the x86 architecture with the original vmware workstation. *ACM Trans. Comput. Syst.*, 30(4):12:1–12:51, 2012.

[51] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient Out-of-Order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, page 991–1008, Baltimore, MD, Aug. 2018. USENIX Association.

[52] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo. SEUSS: skip redundant paths to make serverless fast. In A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic, and M. I. Seltzer, editors, *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pages 32:1–32:15. ACM, 2020.

[53] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A systematic evaluation of transient execution attacks and defenses. In N. Heninger and P. Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 249–266. USENIX Association, 2019.

[54] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski. The rise of serverless computing. *Commun. ACM*, 62(12):44–54, nov 2019.

[55] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler. Sebs: a serverless benchmark suite for function-as-a-service computing. In K. Zhang, A. Gherbi, N. Venkatasubramanian, and L. Veiga, editors, *Middleware '21: 22nd International Middleware Conference, Québec City, Canada, December 6 - 10, 2021*, pages 64–78. ACM, 2021.

[56] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In J. R. Larus, L. Ceze, and K. Strauss, editors, *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pages 467–481. ACM, 2020.

[57] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[58] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, pages 287–305. ACM, 2017.

[59] Fn Project Website. https://fnproject.io, 2021.

[60] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and

K. Winstein. Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, Mar. 2017. USENIX Association.

[61] Y. Fu, L. Xue, Y. Huang, A. Brabete, D. Ustiugov, Y. Patel, and L. Mai. Serverlessllm: Locality-enhanced serverless inference for large language models. *CoRR*, abs/2401.14351, 2024.

[62] M. Golec, R. Ozturac, Z. Pooranian, S. S. Gill, and R. Buyya. ifaasbus: A security- and privacy-based lightweight framework for serverless computing using iot and machine learning. *IEEE Transactions on Industrial Informatics*, 18(5):3522–3529, 2022.

[63] D. Goltzsche, M. Nieke, T. Knauth, and R. Kapitza. Acctee: A webassembly-based two-way sandbox for trusted resource accounting. In *Proceedings of the 20th International Middleware Conference, Middleware 2019, Davis, CA, USA, December 9-13, 2019*, pages 123–135. ACM, 2019.

[64] Google. Collection of sample apps showcasing popular use cases using Cloud Functions for Firebase. https://github.com/firebase/functions-samples, 2022.

[65] Google. Google serverless computing. https://cloud.google.com/serverless, 2022.

[66] J. Gu, X. Wu, W. Li, N. Liu, Z. Mi, Y. Xia, and H. Chen. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 401–417, July 2020.

[67] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, may 2009.

[68] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless computation with openlambda. In A. Clements and T. Condie, editors, *8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, June 20-21, 2016*. USENIX Association, 2016.

[69] A. V. Hof and J. Nieh. BlackBox: A container security monitor for protecting containers on untrusted operating systems. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 683–700, Carlsbad, CA, July 2022. USENIX Association.

[70] B. Holmes, J. Waterman, and D. Williams. Severifast: Minimizing the root of trust for fast startup of sev microvms. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*, La Jolla, CA, USA, 2024. ACM.

[71] Z. Hua, Y. Yu, J. Gu, Y. Xia, H. Chen, and B. Zang. Tz-container: protecting container from untrusted OS with ARM trustzone. *Sci. China Inf. Sci.*, 64(9), 2021.

[72] Huawei. Huawei clound functions. https://developer.huawei.com/consumer/en/agconnect/cloud-function/, 2022.

[73] G. D. H. Hunt, R. Pai, M. V. Le, H. Jamjoom, S. Bhattiprolu, R. Boivie, L. Dufour, B. Frey, M. Kapur, K. A. Goldman, R. Grimm, J. Janakirman, J. M. Ludden, P. Mackerras, C. May, E. R. Palmer, B. B. Rao, L. Roy, W. A. Starke, J. Stuecheli, E. Valdez, and W. Voigt. Confidential computing for openpower. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 294–310, New York, NY, USA, 2021. Association for Computing Machinery.

[74] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM Trans. Comput. Syst.*, 35(4), dec 2018.

[75] IBM. IBM Cloud Functions. https://www.ibm.com/cloud/functions, 2022.

[76] IBM. Serverless computing in financial services. https://www.ibm.com/downloads/cas/Q93RMQMN, 2022.

[77] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 143–157. IEEE Computer Society, 2012.

[78] Z. Jia and E. Witchel. Boki: Stateful serverless computing with shared logs. In R. van Renesse and N. Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 691–707. ACM, 2021.

[79] Z. Jia and E. Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In T. Sherwood, E. D. Berger, and C. Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 152–166. ACM, 2021.

[80] A. Joosen, A. Hassan, M. Asenov, R. Singh, L. N. Darlow, J. Wang, and A. Barker. How does it function?: Characterizing long-term trends in production serverless workloads. In *Proceedings of the 2023 ACM Symposium on Cloud Computing, SoCC 2023, Santa Cruz, CA, USA, 30 October 2023 - 1 November 2023*, pages 443–458. ACM, 2023.

[81] J. Kim and K. Lee. Functionbench: A suite of workloads for serverless cloud function service. In E. Bertino, C. K. Chang, P. Chen, E. Damiani, M. Goul, and K. Oyama, editors, *12th IEEE International Conference on Cloud Computing, CLOUD 2019, Milan, Italy, July 8-13, 2019*, pages 502–504. IEEE, 2019.

[82] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 207–220, New York, NY, USA, 2009. Association for Computing Machinery.

[83] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[84] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, c. Teodorescu, C. Răducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu, and F. Huici. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 376–394, New

York, NY, USA, 2021. Association for Computing Machinery.

[85] A. Kumari, R. K. Behera, B. Sahoo, and S. Misra. Role of serverless computing in healthcare systems: Case studies. In *Computational Science and Its Applications – ICCSA 2022 Workshops: Malaga, Spain, July 4–7, 2022, Proceedings, Part IV*, page 123–134, Berlin, Heidelberg, 2022. Springer-Verlag.

[86] D. Kuvaiskii, D. Stavrakakis, K. Qin, C. Xing, P. Bhatotia, and M. Vij. Gramine-tdx: A lightweight OS kernel for confidential vms. In B. Luo, X. Liao, J. Xu, E. Kirda, and D. Lie, editors, *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, pages 4598–4612. ACM, 2024.

[87] H. Lefeuvre, G. Gain, V. Badoiu, D. Dinca, V. Schiller, C. Raiciu, F. Huici, and P. Olivier. Loupe: Driving the development of OS compatibility layers. In R. Gupta, N. B. Abu-Ghazaleh, M. Musuvathi, and D. Tsafrir, editors, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, pages 249–267. ACM, 2024.

[88] D. Li, Z. Mi, Y. Xia, B. Zang, H. Chen, and H. Guan. Twinvisor: Hardware-isolated confidential virtual machines for ARM. In R. van Renesse and N. Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 638–654. ACM, 2021.

[89] M. Li, L. Wilke, J. Wichelmann, T. Eisenbarth, R. Teodorescu, and Y. Zhang. A systematic look at ciphertext side channels on AMD SEV-SNP. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 337–351. IEEE, 2022.

[90] M. Li, Y. Xia, and H. Chen. Confidential serverless made efficient with plug-in enclaves. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021*, pages 306–318. IEEE, 2021.

[91] M. Li, Y. Zhang, H. Wang, K. Li, and Y. Cheng. CIPHERLEAKS: breaking constant-time cryptography on AMD SEV via the ciphertext side channel. In M. Bailey and R. Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 717–732. USENIX Association, 2021.

[92] S. Li, J. S. Koh, and J. Nieh. Protecting cloud virtual machines from hypervisor and host operating system exploits. In N. Heninger and P. Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1357–1374. USENIX Association, 2019.

[93] Z. Li, J. Cheng, Q. Chen, E. Guan, Z. Bian, Y. Tao, B. Zha, Q. Wang, W. Han, and M. Guo. Rund: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In J. Schindler and N. Zilberman, editors, *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 53–68. USENIX Association, 2022.

[94] Z. Li, L. Guo, Q. Chen, J. Cheng, C. Xu, D. Zeng, Z. Song, T. Ma, Y. Yang, C. Li, and M. Guo. Help rather than recycle: Alleviating cold startup in serverless computing through inter-function container sharing. In J. Schindler and N. Zilberman, editors, *2022 USENIX Annual Technical Conference, USENIX ATC 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 69–84. USENIX Association, 2022.

[95] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou. A measurement study on linux container security: Attacks and countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, pages 418–429. ACM, 2018.

[96] J. Lind, O. Naor, I. Eyal, F. Kelbert, E. G. Sirer, and P. Pietzuch. Teechain: A secure payment network with asynchronous blockchain access. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 63–79, New York, NY, USA, 2019. Association for Computing Machinery.

[97] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, and R. Strackx. Meltdown: Reading kernel memory from user space. *Commun. ACM*, 63(6):46–56, may 2020.

[98] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[99] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni. Faasdom: a benchmark suite for serverless computing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, DEBS '20, page 73–84, New York, NY, USA, 2020. Association for Computing Machinery.

[100] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery.

[101] O. Matei, R. Erdei, A. Moga, and R. Heb. A serverless architecture for a wearable face recognition application. In *Pattern Recognition. ICPR International Workshops and Challenges: Virtual Event, January 10-15, 2021, Proceedings, Part VII*, page 642–655, Berlin, Heidelberg, 2021. Springer-Verlag.

[102] Microsoft. Azure functions. https://azure.microsoft.com/en-us/services/functions/, 2022.

[103] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association.

[104] D. B. Pons, M. S. Artigas, G. París, P. Sutra, and P. G. López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference, Middleware 2019, Davis, CA, USA, December 9-13, 2019*, pages 41–54. ACM, 2019.

[105] J. Powell. Amd sev-snp attestation: Establishing trust

in guests. https://www.amd.com/content/dam/amd/en/documents/developer/lss-snp-attestation.pdf.

[106] W. Qiang, Z. Dong, and H. Jin. Se-lambda: Securing privacy-sensitive serverless applications using SGX enclave. In R. Beyah, B. Chang, Y. Li, and S. Zhu, editors, *Security and Privacy in Communication Networks - 14th International Conference, SecureComm 2018, Singapore, August 8-10, 2018, Proceedings, Part I*, volume 254 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 451–470. Springer, 2018.

[107] A. Qumranet, Y. Qumranet, D. Qumranet, U. Qumranet, and A. Liguori. Kvm: The linux virtual machine monitor. *Proceedings Linux Symposium*, 15, 01 2007.

[108] M. U. Sardar, S. Musaev, and C. Fetzer. Demystifying attestation in intel trust domain extensions via formal verification. *IEEE Access*, 9:83067–83079, 2021.

[109] D. Saxena, T. Ji, A. Singhvi, J. Khalid, and A. Akella. Memory deduplication for serverless computing with medes. In Y. Bromberg, A. Kermarrec, and C. Kozyrakis, editors, *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, pages 714–729. ACM, 2022.

[110] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In A. Gavrilovska and E. Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pages 205–218. USENIX Association, 2020.

[111] M. Shahrad, R. Fonseca, I. n. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.

[112] Z. Shen, Z. Sun, G.-E. Sela, E. Bagdasaryan, C. Delimitrou, R. Van Renesse, and H. Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 121–135, New York, NY, USA, 2019. Association for Computing Machinery.

[113] W. Shin, W.-H. Kim, and C. Min. Fireworks: A fast, efficient, and safe serverless framework using vm-level post-jit snapshot. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 663–677, New York, NY, USA, 2022. Association for Computing Machinery.

[114] L. Soares and M. Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 33–46, Berkeley, CA, USA, 2010. USENIX Association.

[115] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, page 275–287, New York, NY, USA, 2007. Association for Computing Machinery.

[116] R. Spreitzer, F. Kirchengast, D. Gruss, and S. Mangard. Procharvester: Fully automated analysis of procfs side-channel leaks on android. In J. Kim, G. Ahn, S. Kim, Y. Kim, J. López, and T. Kim, editors, *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*, pages 749–763. ACM, 2018.

[117] S. Sultan, I. Ahmad, and T. Dimitriou. Container security: Issues, challenges, and the road ahead. *IEEE Access*, 7:52976–52996, 2019.

[118] B. Trach, O. Oleksenko, F. Gregor, P. Bhatotia, and C. Fetzer. Clemmys: towards secure remote execution in faas. In M. Hershcovitch, A. Goel, and A. Morrison, editors, *Proceedings of the 12th ACM International Conference on Systems and Storage, SYSTOR 2019, Haifa, Israel, June 3-5, 2019*, pages 44–54. ACM, 2019.

[119] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In T. Sherwood, E. D. Berger, and C. Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 559–572. ACM, 2021.

[120] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. M. Swift. Peeking behind the curtains of serverless platforms. In H. S. Gunawi and B. C. Reed, editors, *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 133–146. USENIX Association, 2018.

[121] X. Wei, F. Lu, T. Wang, J. Gu, Y. Yang, R. Chen, and H. Chen. No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 497–517, Boston, MA, July 2023. USENIX Association.

[122] F. Wu, M. Dong, G. Mo, and H. Chen. Treesls: A whole-system persistent microkernel with tree-structured state checkpoint on nvm. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 1–16, New York, NY, USA, 2023. Association for Computing Machinery.

[123] N. Yang, W. Shen, J. Li, Y. Yang, K. Lu, J. Xiao, T. Zhou, C. Qin, W. Yu, J. Ma, and K. Ren. Demons in the shared kernel: Abstract resource attacks against os-level virtualization. In Y. Kim, J. Kim, G. Vigna, and E. Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 764–778. ACM, 2021.

[124] V. Young, P. J. Nair, and M. K. Qureshi. Deuce: Write-efficient encryption for non-volatile memories. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 33–44, New York, NY, USA,

2015. Association for Computing Machinery.

[125] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen. Characterizing serverless platforms with serverlessbench. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 30–44, New York, NY, USA, 2020. Association for Computing Machinery.

[126] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 990–1003, New York, NY, USA, 2014. Association for Computing Machinery.

[127] S. Zhao, P. Xu, G. Chen, M. Zhang, Y. Zhang, and Z. Lin. Reusable enclaves for confidential serverless computing. In J. A. Calandrino and C. Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 4015–4032. USENIX Association, 2023.