TDSnap: Enabling Secure Function-as-a-Service with Trusted Domain Snapshots

Seong-Joong Kim
National Security Research Institute
sungjungk@gmail.com

Seungwon Shin
KAIST
claude@kaist.ac.kr

Myoungsung You
University of Seoul
famous@uos.ac.kr

Abstract

Function-as-a-Service (FaaS) has become a de facto paradigm for deploying cloud applications, yet it exposes sensitive data to untrusted infrastructures. Running functions on confidential virtual machines (CVMs), such as Intel TDX, provides strong isolation for data but significantly increases cold-start latency due to CVM initialization. This paper presents *TDSnap*, the first TDX-aware function snapshot system that records a verified function state with its TD and restores it on demand. Using snapshot-bounded attestation and TD-aware working-set identification, *TDSnap* preserves TDX isolation while reducing cold-start latency. Evaluation shows that *TDSnap* reduces cold-start latency by up to 40x and matches non-TDX snapshot systems.

1 Introduction

Serverless computing has become a widely adopted model for building and deploying large-scale cloud applications. Among its various abstractions, Function-as-a-Service (FaaS) [27] is the most prevalent, enabling developers to structure complex applications as collections of lightweight *functions* that execute only in response to predefined events (e.g., client requests). Thanks to its fine-grained resource efficiency and a pay-as-you-go cost model, major cloud providers have widely adopted FaaS as a core service model [5, 11, 21].

As FaaS increasingly handles privacy-sensitive workloads, such as face recognition [9, 10, 20] and financial transaction processing [6, 13], the need for strong security isolation for function data has become paramount. Ensuring data confidentiality and integrity in FaaS platforms is particularly challenging under their multi-tenant execution model, where functions from different tenants share the same underlying infrastructure. A malicious function can leak sensitive information from co-located functions, while a compromised host OS may tamper with data from all tenant functions by abusing its privileged control [16, 18, 23, 24, 26]. These threats underscore the necessity of robust isolation mechanisms to protect per-function data.

Recent advances in confidential virtual machines (CVMs) [22] offer a promising foundation for addressing these challenges. For instance, Intel Trust Domain Extensions (TDX) [14] creates an isolated virtual machine, called a Trust Domain (TD), whose memory and vCPU states are cryptographically protected from the rest of the system. Running each function inside a dedicated TD provides strong confidentiality and integrity guarantees for its code and data.

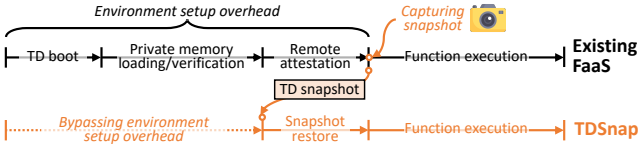


Figure 1: Cold start procedure of existing FaaS and *TDSnap*.

However, this approach introduces severe performance penalties, particularly in the function cold-start, due to the complex bootstrapping process of TDs.

As functions are executed only upon request, running each function inside a dedicated TD requires creating a function TD for every invocation. As shown in Figure 1, this process includes initializing a TD and its metadata, allocating private memory, validating code and data pages, and optionally performing remote attestation. Repeating these steps at every function invocation results in multi-second startup delays; in our measurements, even simple functions incur more than 4 seconds of startup time. Because practical FaaS platforms require sub-second cold start latency, this overhead renders naïve TDX integration impractical.

Previous studies [25, 29] have attempted to mitigate these costs by maintaining a shared, long-lived TD that hosts multiple functions. Although this design eliminates the overhead associated with TD initialization, it sacrifices security isolation: functions within the same shared TD are isolated by software-level protection (e.g., namespaces), leaving them vulnerable to cross-function attacks. Thus, there remains a fundamental tension between strong isolation and fast startup in confidential FaaS environments.

This paper presents *TDSnap*, a novel TDX-compatible snapshot system that reconciles this tension by providing both hardware-backed per-function isolation and fast startup. As shown in Figure 1, *TDSnap* securely records the state of a fully initialized and verified function and its TD as a TD snapshot, which can be restored on demand. During snapshot creation, *TDSnap* generates a TD template that captures immutable TD states for rapid re-boot and identifies working-set pages using a TD-aware page-tracking mechanism. The resulting snapshot is cryptographically protected by TDX and retained in an in-memory cache for fast restore. Upon invocation, *TDSnap* re-boots the TD by reusing the TD template, prefetches the working-set pages into private memory, and verifies integrity and freshness through snapshot-bound attestation. This design preserves TDX's isolation guarantees for each function while substantially reducing cold-start latency.

Contributions. We make the following contributions:

- We identify key performance and security challenges in integrating snapshots with TDX to reduce function cold-start latency.
- We design and implement the first TDX-aware function snapshot system that securely captures and restores functions and their TD states, including verified working-sets.

This work is licensed under a Creative Commons Attribution 4.0 International License.
DAC '26, July 26–29, 2026, Long Beach, CA, USA
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2254-7/2026/07
https://doi.org/10.1145/3770743.3804055

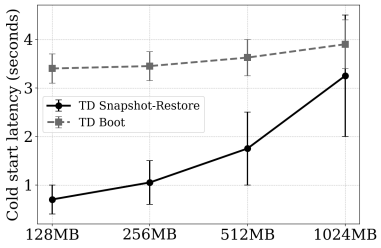


Figure 2: Cold start latency of a full TD boot and a snapshot.

- We demonstrate that *TDSnap* reduces cold-start latency for TDX by up to 40× compared to the baseline and achieves performance comparable to state-of-the-art non-TDX snapshot systems [4, 31].

2 Background and Motivation

2.1 FaaS and Intel TDX

Function-as-a-Service (FaaS). FaaS [6] is a representative paradigm of serverless computing that enables developers to compose complex applications from lightweight functions. Each function is registered to a specific event (e.g., a client request). When an event occurs, the FaaS platform creates an isolated runtime (e.g., a microVM) and loads the function binary and required libraries into it. This process, referred to as a *cold start*, denotes the setup phase in which a function is launched from scratch. The function then processes the incoming request, and upon completion, both the function instance and its allocated resources are entirely released.

Intel TDX. Intel Trust Domain Extensions (TDX) [14] is one of the most widely adopted CVM technologies in modern cloud infrastructures. It creates hardware-isolated virtual machines, called Trust Domains (TDs), protecting TDs from the rest of the system, including the host OS and VMM. A TDX module, a privileged firmware component executing in Secure Arbitration Mode (SEAM), manages the lifecycle of TDs within the server. It exposes a secure interface, known as *SEAMCALL*, through which the VMM interacts with it to perform resource scheduling and exception handling for TDs.

To create a new TD, the VMM issues a sequence of SEAMCALLs that allocate TD structures and configure its private memory. The TDX module measures and validates each page before admitting it into the private memory, ensuring that only verified code and data are loaded. All private-memory pages are protected with hardware-enforced encryption and authentication, preventing even the host OS or VMM from accessing them. Guest physical address (GPA) to host physical address (HPA) translations for this memory are managed by the Secure Extended Page Table (SEPT). Although the VMM may request SEPT updates via SEAMCALLs, every update must be explicitly validated and approved by the TD.

2.2 Integrating TDX into FaaS

In FaaS environments, multiple functions from different tenants are deployed on the same machine, sharing underlying hardware resources. This co-location introduces two major security risks. First, a malicious function can exploit shared resources (e.g., kernel exploit [19]) to access sensitive data of other functions. Second, a compromised privileged software (e.g., the host OS) can manipulate the code and data of all tenant functions by abusing its privileges. Such threats are severe for data-sensitive workloads, including facial recognition and financial processing.

Isolating each function within a dedicated TD prevents its sensitive data from leaking to or being tampered with by malicious functions or system software. However, integrating TDX into FaaS introduces significant performance challenges due to both the complexity of TDX operations and the transient nature of functions, especially during function cold-starts. When a request arrives, a new TD must be created, and the function binary along with its dependent libraries must be loaded into the TD's private memory, as shown in Figure 1. During this process, every page is measured and validated by the TDX module, substantially increasing cold-start latency. As shown by the TD-Boot plot in Figure 2, even a small function (128 MB) requires more than three seconds to launch, while a 1 GB function incurs over four seconds. Considering that typical FaaS platforms demand sub-second startup latency, such delay is unacceptable. Our analysis reveals the root cause: for each 4 KB page, the VMM must issue at least two SEAMCALLs to populate and measure the page, meaning a 1 GB function requires roughly 524K SEAMCALL invocations (100K CPU cycles per page).

Most prior studies [25, 29] mitigate the cold-start latency of CVM-backed FaaS by relying on CVM-sharing and warm-start techniques. CoFunc [29], for instance, maintains a long-lived shared TD that dynamically hosts multiple functions. Instead of creating a fresh TD per invocation, functions are loaded into this shared TD, and verified library pages already resident in the TD are reused across functions through copy-on-write (CoW) techniques.

However, such sharing-based designs have fundamental limitations that stem directly from collapsing hardware-isolated trust boundaries. Because multiple tenants execute within the same TD, isolation is enforced only through in-guest software mechanisms (e.g., namespaces) rather than TDX's hardware-based isolation. A malicious or compromised function can therefore access sensitive data of other co-located functions. Thus, a breach in one function compromises all others within that shared TD. Also, the long-lived nature of the shared TD allows attacker-controlled code to persist across invocations, enabling stealthy long-term attacks. For functions carrying sensitive data or requiring strict security policies, such trust collapse renders CVM-sharing inherently inappropriate. From a performance perspective, CVM-sharing also presents structural inefficiencies. CoW is beneficial only when tenants share identical pages. When functions require distinct libraries, CoW opportunities collapse and large amounts of private memory must be imported, incurring substantial TDX overhead. Also, if no suitably *warm* shared TD exists, a new TD should be created from scratch. Both cases lead to high startup latency (see Section 4).

2.3 Challenges in Snapshotting for TDX

In contrast to CVM-sharing approaches, this work introduces a snapshot-based mechanism for TDX that mitigates cold-start latency while preserving hardware-enforced isolation between functions. Snapshotting has been widely employed in FaaS systems [4, 8, 31] to reduce startup overhead by capturing the fully initialized state of a function into a restorable image. Applied to TDX, our system first boots a dedicated TD for a target function, loads the

function binary and its dependencies into the TD's private memory, and verifies all loaded pages through the TDX measurement process. It then records the TD's complete state, including verified private pages, into a TD snapshot file. Upon invocations, a TD can be rapidly restored from the snapshot, bypassing the costly full TD boot and private memory initialization. Despite its clear potential, snapshots for TDX remain unexplored. We identify two challenges that must be addressed to enable function snapshots for TDX.

Lack of working-set awareness. A snapshot preserves the full memory image of a fully initialized function, which includes both cold pages used only during initialization and working set pages frequently accessed during request execution. To achieve low startup overhead, a snapshot system must prefetch only the working set while lazily loading cold pages; otherwise, restoration latency grows substantially [4, 8, 16, 30, 31]. However, working set identification in existing snapshot systems relies on host visibility into guest memory access behavior, an assumption that does not hold in TDX. The TDX threat model prevents the host from inspecting private pages or observing guest level memory events, which eliminates the basic signals these systems depend on. For example, page fault tracing is infeasible because TD page faults are handled internally without VM exits, and FaaSnap's host page cache based introspection [4] cannot operate since TD private pages and SEPT metadata are inaccessible to the host. These constraints are intrinsic to TDX rather than limitations of specific mechanisms; detailed discussion is provided in Section 5. As a result, no existing approach can determine a function's working set inside a TD, forcing snapshot restoration to reload all pages and thereby inflating cold-start latency.

To show this problem, we implement a baseline TD snapshot mechanism that restores all pages from a snapshot into a TD and then measure its cold-start latency. As shown by the *TD snapshot* plot in Figure 2, a snapshot offers notable benefits for small functions. For example, with a 128 MB footprint, cold-start latency drops below one second. However, even a 256 MB function already exceeds this threshold, and the advantage diminishes rapidly as memory size increases; when the footprint reaches 1 GB, startup latency exceeds three seconds, and the tail latency surpasses that of full TD boot. This overhead stems from the expensive private memory loading and verification steps of TDX, which involve multiple SEAMCALLs. Thus, reducing the number of pages reloaded during cold start is essential for making TD snapshots practical.

Lack of snapshot security mechanisms. Applying function snapshots in TDX environments introduces a security challenge that prior non TDX snapshot systems [4, 8, 31] do not consider. In FaaS, the FaaS platform is responsible for restoring snapshots during cold starts, and because the platform operates on the host side, the snapshot containing the TD state must also be stored on the host side (e.g., a file system). Under the TDX threat model, however, the host OS cannot be trusted, which exposes the stored snapshot to several risks. An attacker with host access can read snapshot contents, revealing private memory pages that may contain sensitive data such as credentials captured during snapshot creation. The attacker can also mount rollback attacks by replacing a recent snapshot with an older version containing a known vulnerability, thus causing the TD to run an outdated binary while appearing legitimate. Since no prior work provides security mechanisms for TD snapshots, there is a clear need for a TDX compatible snapshot system that not only preserves confidentiality and integrity but also detects snapshot tampering and verifies that the restored state corresponds to the originally attested TD.

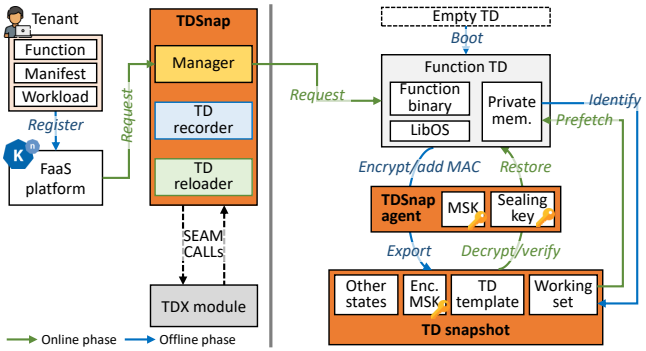


Figure 3: The overall architecture of *TDSnap*.

3 *TDSnap* Design

3.1 Threat Model

Similar to previous studies in CVMs [12, 17], we trust the TDX module and Intel-provisioned service TDs [1] that the platform uses to support TDX functionality. We assume that an attacker can (1) compromise the privileged software (e.g., the host OS) and (2) deploy a malicious function. The attacker's goal is to inspect or modify the private memory state or vCPU registers of other functions. The security goal of *TDSnap* is to ensure that each function's TD maintains confidentiality and integrity of its private memory against a compromised host or other functions. We exclude attacks that exploit known vulnerabilities in a function binary itself, as TDX does not aim to mitigate bugs within the guest software [7].

3.2 System Architecture

TDSnap implements a TDX-compliant snapshot mechanism that relies solely on SEAMCALLs. Specifically, it repurposes the TD migration feature [2], originally designed for live TD migration, to support secure snapshot creation and restoration. As shown in Figure 3, the workflow consists of an offline phase (blue arrows) and an online phase (green arrows). In the offline phase, *TDSnap* exports a function TD's execution state into a snapshot file and performs TD-aware page tracking to construct working-set pages. The generated snapshot is then encrypted and authenticated before being stored on the host file system. In the online phase, *TDSnap* restores the snapshot, prefetches the working-set pages, and performs snapshot-bound attestation to verify the integrity and freshness of the reconstructed TD. The function can handle requests immediately, while missing pages are fetched on demand by *TDSnap*. This design significantly reduces cold-start latency while preserving per-function isolation and full compatibility with TDX.

3.3 Offline Phase: TD Snapshot Creation

In the offline phase, a tenant registers a function by providing its binary, manifest, and a sample workload. The TD recorder is responsible for generating a TD snapshot for each registered function. As shown in Figure 3, it boots a fresh TD with our TDX-compatible LibOS [17] and loads the function binary and libraries according to

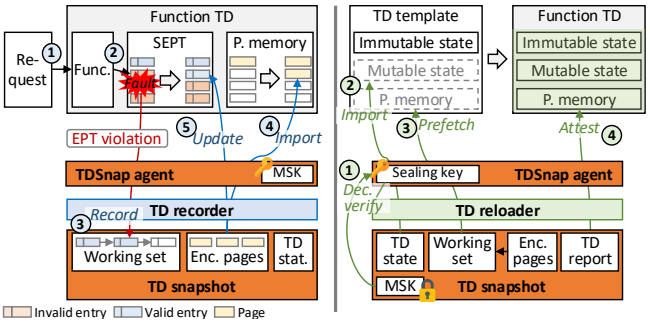


Figure 4: The (S)EPT violation tracking mechanism (left) and snapshot-based function TD restoration (right).

the manifest, which specifies the function's required runtime configuration (e.g., memory layout), thereby populating the TD's private memory and SEPT with code and data pages. The TD recorder may optionally perform remote attestation to confirm the integrity of this initialized environment. Next, the Manager creates a verified *TDSnap* agent that operates as a service TD [1] to handle TD migration interfaces. Through the TDX module, this agent obtains controlled access to the function TD's private memory and derives a Migration Session Key (MSK). The TD recorder then suspends the function TD and exports its private pages via EXPORT SEAMCALLs [2]. As shown in Figure 3, the agent encrypts each page and computes a MAC using the MSK before writing them to the host file system, producing an initial TD snapshot that securely captures both the TD's immutable state and its private memory.

Working-set identification. Unlike previous snapshot systems, *TDSnap* introduces a working-set identification mechanism that operates fully within TDX constraints by exploiting architectural properties unique to TD migration. Under normal execution, page faults inside a function TD do not trigger TD exits and are resolved internally, leaving the VMM unable to observe memory access patterns. However, during TD migration, an EXPORT SEAMCALL flushes all SEPT entries, and the first access to any private page subsequently triggers an (S)EPT violation. Unlike ordinary page faults, these violations necessarily cause VM exits, allowing the VMM to observe the faulting GPA while still preventing access to the page contents. This behavior provides a unique and TDX-compliant visibility channel that forms the basis of our design. Figure 4 illustrates the process. ① The TD recorder initiates tenant-provided workloads, allowing the function TD to process a request. ② On each first access after a SEPT flush, the TD raises an (S)EPT violation, which results in a VM exit. ③ The recorder logs the GPA from the exit context, retrieves the corresponding page from the initial snapshot, and invokes an IMPORT SEAMCALL. ④ The migration agent decrypts and imports the page, and ⑤ updates the SEPT entry accordingly. After completing the request, the TD recorder flushes SEPT entries and repeats the process, aggregating pages consistently accessed across iterations into the working-set.

Snapshot sealing. After profiling the working-set, the TD recorder exports the remaining TD mutable state (e.g., vCPU registers) into the TD snapshot. Because all snapshot contents are encrypted with the MSK, this key must also be stored alongside the snapshot for future restoration. However, the MSK is intended solely for TD migration and is not designed to be persistently stored in the untrusted host file system. To securely store the MSK, *TDSnap* leverages TDX's built-in key-management mechanism. The agent obtains a sealing key from the TDX module; this key is cryptographically bound to the agent's identity and is never exposed outside the processor. The agent encrypts the MSK with the sealing key and embeds the sealed MSK in the snapshot header. As only the same attested agent can later derive the sealing key, we ensure that no other entity, including the host OS, can decrypt or use the snapshot. As the output of the offline phase, *TDSnap* produces a TD snapshot that captures a function TD's complete execution state. The snapshot consists of the encrypted MSK, TD mutable state, the working-set, private-memory pages, and the attestation report. Among them, the working-set is retained in a per-function in-memory cache to support fast prefetching. The remaining immutable state (e.g., the SEPT root and TDVPS), which does not change after TD creation, is also preserved as a TD template in memory for fast TD re-boot.

3.4 Online Phase: TD Snapshot Restore

In the online phase, the FaaS platform handles function invocations by restoring TD snapshots. Upon receiving a request, the platform forwards it to the Manager, which temporarily holds the request and initiates the restoration executed by the TD reloader. It first locates the corresponding snapshot and instructs the agent. As shown in the right part of Figure 4, the agent obtains the sealing key and then ① decrypts the snapshot's MSK to validate both the integrity digest and the freshness tag derived from a hardware monotonic counter, preventing rollback attacks and ensuring that the snapshot has not been modified. The reloader then retrieves the corresponding TD template and ② populates it with the TD mutable state extracted from the snapshot, avoiding the overhead of re-creating immutable TD states. It subsequently ③ prefetches the working-set pages into private memory using IMPORT SEAMCALLs. For each page, the agent decrypts the content and verifies its MAC via the MSK before importing it. Because the working-set is maintained in an in-memory cache, *TDSnap* avoids disk I/O during prefetching. Note that pages decrypted with the MSK are never stored in plaintext inside the TD. When written into private memory, the hardware engine re-encrypts them with the TD's private key.

Snapshot-bounded attestation. Once the working-set is restored, *TDSnap* performs snapshot-bounded attestation to ensure that the reconstructed TD matches the attested state captured during snapshot generation. The agent retrieves the verified hash-based tag of the snapshot, which reflects the TD's current state (e.g., TDCS and MRTD/RTMR values). As shown by ④, it acts as a proxy attester and verifies that the TDREPORT matches the TD identity and configuration recorded at snapshot creation. Because the snapshot remains immutable, and all pages are MAC-protected under the MSK, the agent also authenticates each imported page. This mechanism ensures that without re-executing the full boot sequence, the attestation report reflects a previously verified software stack, maintaining trust guarantees for the function owner.

On-demand paging. After these steps, the TD template becomes a fully restored and verified function TD, and the Manager forwards the pending request to it. During execution, any access to a non-prefetched page triggers a controlled on-demand fault, which the TD reloader resolves by securely importing the missing page.

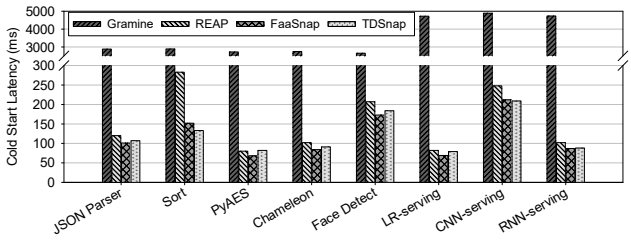


Figure 5: Cold-start latency of each function.

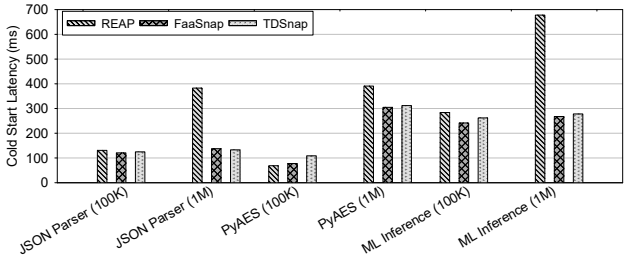


Figure 6: Cold-start latency variation on different input sizes.

Once the request completes, the Manager flushes the function TD's mutable state and converts it back into a TD template.

4 Evaluation

4.1 Evaluation environment

We implement a full prototype integrated with Knative and extend QEMU to support TDX migration. *TDSnap*'s Manager, TDRecorder, and TDLoader are implemented as VMM-level plugins, while the *TDSnap* agent is built using Intel's Migration TD [1]. All experiments are conducted in a real-world FaaS environment using two TDX-enabled servers (Xeon 6544Y with TDX 1.5). Guest TDs are provisioned with 4 vCPUs and 1 GB memory, matching typical FaaS settings. For workloads, we select eight functions from FunctionBench [15], each packaged as a Gramine-TDX manifest. We compare *TDSnap* against two baselines: vanilla Gramine-TDX [17] as a lower-bound baseline without TD snapshots, and state-of-the-art non-CVM snapshot systems (REAP [31] and FaaSnap [4]) as upper-bound baselines without the overhead of TDX.

4.2 End-to-end performance

We first measure the cold-start latency of each selected function. As shown in Figure 5, *TDSnap* achieves 121 ms for simple functions (e.g., JSON parser) and 209 ms for complex functions (e.g., RNN serving) on average, showing an order-of-magnitude lower latency than Gramine-TDX. This improvement stems from our optimized snapshot-restore, which combines TD templates with working-set prefetching. Compared to REAP and FaaSnap, *TDSnap* delivers comparable performance despite the overheads imposed by TDX. REAP suffers from disk I/O overheads during working-set prefetching, whereas *TDSnap* avoids such overheads by storing compact working sets in an in-memory cache. FaaSnap mitigates disk I/O through concurrent loading, but this optimization is incompatible with TDX because private-memory imports and SEPT updates must follow strict sequential SEAMCALL operations. *TDSnap* preserves similar performance by maintaining working-set pages in an in-memory cache at low memory cost (see Section 4.3). We also evaluate working-set stability under varying input sizes. Snapshots generated with 100 KB inputs are evaluated with both 100 KB and 1 MB requests. As shown in Figure 6, *TDSnap* maintains performance comparable to non-CVM snapshot systems, showing that *TDSnap* identifies stable working sets across function TD invocations.

Table 1: TD booting latency comparison.

Step	From scratch	TD template
TD creation	355ms	-
SEPT init	263ms	-
Metadata setup	98ms	-
Working set restore	140ms	140ms
Total	856ms	140ms

Comparison with CVM-sharing. We attempted to compare *TDSnap* with CoFunc; however, its open-source artifact does not include a TDX implementation. We therefore rely on the published evaluation results (Section 7.2) from the CoFunc paper [29]. When executing the Face Detect function, CoFunc reports 17 ms latency by reusing a long-lived shared TD and forking required library pages from co-located functions. These optimizations, however, depend on the presence of an already active shared TD populated with reusable library pages. When such a TD is unavailable, either because no functions are running or usable libraries do not overlap, startup latency sharply increases to 657 ms due to the substantial overhead of private-memory loading under TDX. In contrast, *TDSnap* consistently achieves 184 ms latency (3× faster) by restoring the Face Detect function from its TD snapshot, without requiring a persistent shared TD or assuming inter-function library sharing.

4.3 Microbenchmark

TD template. *TDSnap* employs TD templates to eliminate the cost of reconstructing immutable TD state during each invocation. As shown in Table 1, full TD booting incurs substantial overhead from creating the SEPT root and TDCS metadata. In contrast, a TD template preserves these immutable states and restores only mutable states from the template, reducing TD boot latency by 6×. Although maintaining TD immutable state introduces persistent memory usage, the cost is negligible: for a 4-vCPU TD with 256 MB of memory, the TD template occupies only ≈0.14 MB per function.

Working-set size. We next evaluate the impact of our EPT-violation tracking, which enables *TDSnap* to prefetch only the working-set pages rather than restoring the full-memory initial snapshot. Table 2 reports the ratio between the initial snapshot and the extracted working set. For simple functions, *TDSnap* reduces the memory footprint to 3.3–7.5%, indicating that most pages accessed during initialization are unnecessary for request processing and are correctly excluded as cold pages. Even for memory-intensive functions, the working set remains compact at 12–15%, amounting to only a few tens of megabytes (about 30 MB). Maintaining these working sets in an in-memory cache enables fast prefetching with negligible persistent memory overhead. Moreover, *TDSnap* maintains one working-set cache per TD snapshot rather than per function instance, ensuring that memory usage remains stable regardless of the number of concurrently executing function TDs.

Working-set accuracy. The right-most columns of Table 2 present the consistency ratio and the number of on-demand page faults,

Table 2: Working-set ratio (size), consistency, and page-fault statistics. Values in the parentheses are standard deviation.

Category	Function	Working-set ratio	Consist. ratio	Page faults
Simple	JSON parser	3.3% (8.4MB)	98.5%	48 (5)
	Sort	3.6% (9.2MB)	97.2%	69 (7)
	PyAES	4.8% (12.2MB)	96.8%	109 (10)
	Chameleon	4.4% (11.2MB)	97.0%	84 (8)
Complex	Face detect	7.5% (19.2MB)	92.1%	369 (33)
	LR serving	14.4% (36.8MB)	94.6%	244 (38)
	CNN serving	12.9% (33.0MB)	89.5%	402 (24)
	RNN serving	14.5% (37.1MB)	93.8%	328 (49)

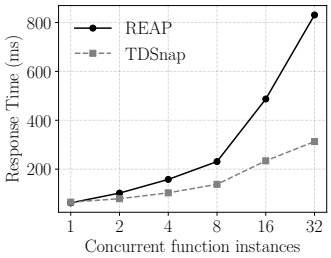


Figure 7: Concurrent function startup latency.

both of which quantify working-set accuracy. On average, *TDSnap* achieves a 97.3% consistency ratio for simple functions and 92.5% for complex functions, demonstrating that EPT-violation tracking reliably identifies pages reused across multiple invocations. To further assess accuracy, we measure on-demand page faults that occur when a function accesses pages not included in its working-set. *TDSnap* incurs fewer than 50 such faults for simple workloads and fewer than 410 for complex ones. Across all benchmarks, more than 95% of accessed pages are included in the working-set. This confirms that *TDSnap* captures the dominant access patterns with high precision, minimizing on-demand faults during execution.

4.4 Concurrent Function Startup

Finally, we evaluate scalability under concurrent cold-starts. Figure 7 shows cold-start latency for parallel initialization of HelloWorld functions. REAP exhibits exponential slowdowns, from 62 ms (1 instance) to 831 ms (32 instances), due to increased disk I/O. In contrast, *TDSnap* scales linearly, increasing only from 65 ms to 313 ms across the same concurrency range. This improvement stems from the use of TD templates, which reduce SEAMCALL invocations during TD booting, and in-memory working sets, which eliminate disk I/Os during snapshot restoration. These results show that *TDSnap* preserves low cold-start latency even under bursty workloads while maintaining TDX's security guarantees.

5 Related Work

Function snapshots. A variety of snapshot systems [4, 8, 30, 31] have been proposed to reduce cold-start latency in FaaS environments. REAP [31] employs a customized page-fault handler to record working-set pages and persist them to the local file system, while FaaSnap [4] refines this approach through locality-aware working-set detection and concurrent prefetching. However, none of them are designed with TDX's threat model in mind. As a result, their working-set identification methods cannot be applied to function TDs. For example, custom fault handlers inside a TD cannot safely export monitored GPAs without introducing custom I/O channels that enlarge the attack surface, nor can these systems protect the resulting snapshots from the untrusted host OS or VMM. In contrast, *TDSnap* is the first system to leverage TDX migration SEAMCALLs to implement a TDX-compliant snapshot mechanism.

CVMs for FaaS. Several recent systems [25, 29] apply TDX or other CVMs to FaaS using CVM-sharing and warm-start techniques, but these designs exhibit fundamental security and performance limitations (see Section 2.2 and Section 4). SEVeriFast [12] improves SEV MicroVM boot latency via a specialized minimal bootstrap path, yet it focuses solely on CVM boot-time optimizations and does not support fast function-initialization mechanisms such as snapshot-based restarts and working-set extraction. Earlier SGX-based systems such as PIE [18], Cryonics [16], and reusable-enclave designs [32] accelerate enclave startup through snapshot or rewind techniques, but their approaches rely on SGX's process-level semantics and therefore do not generalize to TDX's VM-level architecture. In contrast, *TDSnap* is the first system to design a snapshot mechanism that operates entirely within TDX's boundary.

6 Discussions and Limitations

Target functions. *TDSnap* enables fast snapshot-based TD startup with minimal in-memory caching, allowing snapshots to be created for all registered functions if desired. Nevertheless, combining *TDSnap* with warm-start techniques can further optimize resource usage. Production traces [28] show that fewer than 10% of functions are invoked every minute; these highly active functions can justify remaining warm in memory despite the associated memory cost. For the remaining majority of functions, which exhibit normal invocation frequencies, *TDSnap* offers an effective alternative by storing them as TD snapshots and restoring them on demand. This avoids the consistent memory cost of warm instances while still delivering low-latency startup. Finally, very cold functions (about 3.5%), invoked less than once per week, can have their working-set evicted from the in-memory cache and placed on disk.

Function chaining. A promising direction for future work is extending *TDSnap* to support fast inter-TD communication. Although *TDSnap* accelerates individual function startup, FaaS workloads may require low-latency communication among functions to support function chaining. Integrating *TDSnap* with emerging TDX Connect [3] enables secure, high-performance communication channels between function TDs whose state can also be captured within their TD snapshots, eliminating repeated setup overheads and further improving end-to-end performance.

7 Conclusion

This work presents *TDSnap*, the first TDX-compatible snapshot system that enables secure and fast initialization of serverless functions. By leveraging TD migration SEAMCALLs, *TDSnap* supports protected snapshot generation and reliable restoration with accurate working-set identification. Evaluations show that *TDSnap* reduces cold-start latency by up to 40× compared to the baseline.

8 ACKNOWLEDGEMENT

We thank the anonymous reviewers. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government(MSIT) (No. RS-2026-25496281).

References

[1] A rust version of Migration TD. https://github.com/intel/MigTD.

[2] Intel TDX Module Architecture Specification: TD Migration. https://cdrdv2-public.intel.com/733578/intel-tdx-module-1.5-td-migration-spec-348550002.pdf.

[3] Intel® TDX connect architecture specification. https://cdrdv2-public.intel.com/773614/intel-tdx-connect-architecture-specification.pdf.

[4] Lixiang Ao, George Porter, and Geoffrey M. Voelker. FaaSnap: FaaS made fast using snapshot-based VMs. In *EuroSys*, 2022.

[5] AWS. Aws lambda. https://aws.amazon.com/lambda.

[6] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Communications of the ACM*, 62(12):44–54, 2019.

[7] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel tdx demystified: A top-down approach. *ACM Computing Surveys*, 56(9):1–33, 2024.

[8] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, 2020.

[9] Muhammed Golec, Ridvan Ozturac, Zahra Pooranian, Sukhpal Singh Gill, and Rajkumar 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, 2021.

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

[11] Google. Google cloud functions.

[12] Benjamin Holmes, Jason Waterman, and Dan Williams. Severifast: Minimizing the root of trust for fast startup of sev microvms. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1045–1060, 2024.

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

[14] Intel. Intel® trust domain extensions (intel tdx).

[15] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *Proceedings of IEEE International Conference on Cloud Computing (CLOUD '19)*, 2019.

[16] Seong-Joong Kim, Myoungsung You, Byoungjun Kim, and Seungwon Shin. Cryonics: Trustworthy Function-as-a-Service using Snapshot-based Enclaves. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2023.

[17] Dmitrii Kuvaiskii, Dimitrios Stavrakakis, Kailun Qin, Cedric Xing, Pramod Bhatotia, and Mona Vij. Gramine-TDX: A Lightweight OS Kernel for Confidential VMs. In *ACM CCS*, 2024.

[18] Mingyu Li, Yubin Xia, and Haibo Chen. Confidential serverless made efficient with plug-in enclaves. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2021.

[19] Xing Li, Xue Leng, and Yan Chen. Securing serverless computing: Challenges, solutions, and opportunities. *IEEE Network*, 37(2):166–173, 2022.

[20] Oliviu Matei, Rudolf Erdei, Alexandru Moga, and Robert Heb. A serverless architecture for a wearable face recognition application. In *International Conference on Pattern Recognition*, pages 642–655. Springer, 2021.

[21] Microsoft. Azure functions.

[22] Arttu Paju, Muhammad Owais Javed, Juha Nurmi, Juha Savimäki, Brian McGillion, and Billy Bob Brumley. Sok: A systematic review of tee usage for developing trusted applications. In *Proceedings of the 18th International Conference on Availability, Reliability and Security*, pages 1–15, 2023.

[23] Joongun Park, Seunghyo Kang, Sanghyeon Lee, Taehoon Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. Hardware-hardened sandbox enclaves for trusted serverless computing. *ACM Transactions on Architecture and Code Optimization*, 21(1):1–25, 2024.

[24] Maryam Rostamipoor, Seyedhamed Ghavamnia, and Michalis Polychronakis. Leakless: Selective data protection against memory leakage attacks for serverless platforms. In *Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2025.

[25] Patrick Sabanic, Masanori Misono, Teofil Bodea, Julian Pritzi, Michael Hackl, Dimitrios Stavrakakis, and Pramod Bhatotia. Confidential serverless computing. *arXiv preprint arXiv:2504.21518*, 2025.

[26] Martin Schwarzl, Pietro Borrello, Andreas Kogler, Kenton Varda, Thomas Schuster, Michael Schwarz, and Daniel Gruss. Robust and scalable process isolation against spectre in the cloud. In *European Symposium on Research in Computer Security*, pages 167–186. Springer, 2022.

[27] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*, pages 1063–1075, 2019.

[28] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*, pages 205–218, 2020.

[29] Jiacheng Shi, Jinyu Gu, Yubin Xia, and Haibo Chen. Serverless Functions Made Confidential and Efficient with Split Containers. In *USENIX Security*, 2025.

[30] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433, 2020.

[31] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, Analysis, and Optimization of Serverless Function Snapshots. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[32] Shixuan Zhao, Pinshen Xu, Guoxing Chen, Mengya Zhang, Yinqian Zhang, and Zhiqiang Lin. Reusable enclaves for confidential serverless computing. In *32nd USENIX security symposium (USENIX Security 23)*, pages 4015–4032, 2023.