TeeMate: Fast and Efficient Confidential Container using Shared Enclave

Chulmin Lee[†]
Seoul National University
2harry@snu.ac.kr

Jaewon Hur[†]
Seoul National University
hurjaewon@snu.ac.kr

Sangho Lee
Microsoft Research
Sangho.Lee@microsoft.com

Byoungyoung Lee
Seoul National University
byoungyoung@snu.ac.kr

Abstract

Confidential container is becoming increasingly popular as it meets both needs for efficient resource management by cloud providers, and data protection by cloud users. Specifically, confidential containers integrate the container and the enclave, aiming to inherit the design-wise advantages of both—i.e., resource management and data protection. However, current confidential containers suffer from large performance overheads caused by i) a larger startup latency due to the enclave creation, and ii) a larger memory footprint due to the non-shareable characteristics of enclave memory.

This paper explores a design conundrum of confidential container, examining why the confidential containers impose such large performance overheads. Surprisingly, we found there is a universal misconception that an enclave can only be used by a single (containerized) process that created it. However, an enclave can be shared across multiple processes, because an enclave is merely a set of physical resources while the process is an abstraction constructed by the host kernel.

To this end, we introduce TeeMate, a new approach to utilize the enclaves on the host system. Especially, TeeMate designs the primitives to i) share the enclave memory between processes, thus preserving memory abstraction, and ii) assign the threads in enclave between processes, thus preserving thread abstraction. We concretized TeeMate on Intel SGX, and implemented confidential serverless computing and confidential database on top of TeeMate based confidential containers. The evaluation clearly demonstrated the strong practical impact of TeeMate by achieving at least 4.5 times lower latency and 2.8 times lower memory usage compared to the applications built on the conventional confidential containers.

1 Introduction

Cloud computing offers several advantages in resource management, allowing its users to focus on their application development without the burdens of managing computing resources [10, 12–14, 39, 52]. Specifically, cloud providers take the complete charge of managing the entire system resources (e.g., CPU, memory, and storage) on which cloud users easily run their application. Due to these advantages, emerging cloud service models such as Software as a Service (SaaS) [13, 52] and Kubernetes as a Service (KaaS) [14] have gained the strong popularity.

Looking into the technical aspect of these cloud service models, *container* [26, 40] (or OS-level virtualization) plays the key role as it facilitates both resource management and isolation. To be specific, each container is assigned with isolated resources, allowing the cloud providers to manage the resources per process, avoiding costly full virtualization [21]. This is enabled by two key features provided by the underlying OS: i) namespace [41], which provides a different userland view over the resources per process (e.g., files and network), and ii) cgroup [22], which limits the CPU and memory usage per process. These two features construct a containerized environment, which serves as a basic management unit by the cloud providers.

Meanwhile, confidential computing [2, 7, 11, 16, 27] has gained strong popularity in clouds, as cloud users demand strong security guarantees over their data. Especially, there is a growing need to exclude the cloud providers from trusted path, as the cloud handles a large amount of privacy-sensitive data that can be in conflict of interest. To meet such security demands, confidential computing (including Intel SGX [16], AMD SEV [2], and Intel TDX [33]) introduces an *enclave*, which is a trusted execution environment. Specifically, the enclave is protected from all systems components including operating systems, hypervisors, and even the other enclaves [2, 65] such that the enclave owner can safely execute their workloads without trusting the cloud providers.

In this paper, we explore a design conundrum of *confidential containers* [60], which integrates aforementioned two techniques, containers and confidential computing. To be specific, confidential containers aim at inheriting the design-wise advantages of each—i.e., resource management capability from containers and data protection capability from confidential computing. As it is naturally thought, current confidential containers construct a single container with a single enclave, which serves as a basic management unit

† These authors contributed equally to this work.

by the host system. For example, SGX enclaves are already widely used with the containers to construct a protected environment in the container [16].

However, current confidential containers suffer from large performance overheads. Especially, we found that integrating the enclave into the container incurs i) a larger startup latency due to the inherent security mechanism when creating the enclaves [79, 92], and ii) a larger memory footprint due to the non-shareable characteristics of enclave memory [66, 79, 93]. We confirmed these overheads and root causes by conducting preliminary experiments on two popular cloud applications, i.e., serverless computing [5, 10, 12], and database [47, 49].

Surprisingly, after the analysis, we found there is a universal misconception that a single enclave must be dedicated to only a single process. In other words, all the previous works, which use the enclaves [57, 60, 61, 73, 86, 87, 90] have (incorrectly) assumed the process that initially created the enclave can exclusively own it. However, we found that it is wrong, and an enclave can be shared across multiple processes (and containers) as long as we preserve the memory abstraction and thread abstraction assumed by the host operating systems.

Based on this observation, we design TeeMate, which is a new approach to utilize the enclaves in the host's perspective. In particular, TeeMate enables a single enclave to be shared across multiple processes so that the host kernel can manage (and isolate) them with different containers. Thus, TeeMate successfully solves the issues of startup latency and memory footprint as it avoids creating a new enclave every time, and enables to share the enclave memory between different (containerized) processes on the host kernel.

In order to share the same enclave across different processes, we design TeeMate to i) preserve the memory abstraction on the memory of shared enclave, and ii) preserve the thread abstraction on the enclave's threads. Then, we discuss intra-enclave isolation to guarantee the isolation between the processes using the same enclave. We concretized these concepts in the context of Intel SGX, which is an already widely used confidential computing technology in the cloud [16].

We implemented TeeMate on the secured version of two major cloud applications, i) confidential serverless computing, and ii) confidential database. The evaluation results clearly indicate that TeeMate outperforms applications using current confidential containers in terms of latency and memory usage. Specifically, TeeMate achieves a latency speedup of 4.54-6.98 times and exhibits memory footprint by only 20-36% in confidential serverless computing. Similarly, in confidential database applications, TeeMate achieves a latency speedup of 277.6-1046.6 times and reduces memory footprint up to 41%. Thus, we confirm that TeeMate is a ready to use framework on current cloud infrastructure, which solves the performance issues without any hardware modification.

2 Background

This section explains container (§2.1) and enclave (§2.2), which are the basic unit of resource management and data protection. Then, we introduce confidential container, which integrates both to inherit the design-wise advantages (§2.3).

2.1 Container based Resource Management

Container technologies [26, 40] refer to the software for managing and isolating the system resources using OS primitives [22, 41]. Especially, the containers work at the granularity of a process, as it is the basic unit for organizing the system resources (e.g., virtual address space, CPU registers, and opened files per each process). In particular, containers are implemented using kernel subsystems, i) namespace [41], which provides a different (isolated) userland view over the system resources (e.g., files and network), and ii) cgroup [22], which limits the CPU and memory used by the process. They populate a containerized environment, inside which the process runs as if it has its own computer system.

Cloud providers heavily rely on the containers as they facilitate the resource management and isolation with minimal performance overheads [1, 20]. Specifically, recent trends in cloud industry have triggered the widespread adoption of containers such as serverless computing [10, 12] and microservice architecture [44]. For example, almost 60% of organizations that use the cloud have adopted the container technologies [53]. As another example, Software as a Service (SaaS) model, which also heavily uses the containers, is expected to grow at a CAGR of 13% from 2023 to 2030 [31].

2.2 Enclave based Data Protection

Enclave is the basic unit of protection by confidential computing technologies (e.g., Intel SGX [16], AMD SEV [2], and Intel TDX [33]) that is hardware isolated from the systems components (e.g., operating systems and hypervisors). To be specific, the CPUs construct an enclave by i) populating and encrypting the memory region used by the enclave, and ii) protecting the enclave's register context through access control. More specifically, when creating an enclave, an initial image (i.e., code and data) is copied into the protected memory region, and hash checked so that the enclave owner can ensure the integrity of the loaded image. After that, CPU encrypts the memory and isolates the register context such that any other components including even another enclave cannot access the original one's data, thus guaranteeing the confidentiality.

Thanks to its strong security guarantees, emerging cloud applications use the enclave to protect their data in a potentially compromised cloud environment. Especially, current trends of AI to handle a large amount of privacy-sensitive data have pushed to use the enclaves. In response, several open source projects for confidential computing have been initiated (e.g., [4, 15]) and cloud providers quickly announced the support for confidential computing (e.g., [11, 25]).

2.3 Confidential Container: Intersection of Container and Enclave

Confidential container [15, 60] has been introduced to meet both needs in the cloud industry as explained above. To be specific, confidential container integrates the container and the enclave to inherit the design-wise advantages of both—i.e., efficient resource management by cloud providers, and data protection for cloud users. For example, SGX enclaves are already widely used for the confidential containers as they are originally designed for process level isolation [65]. In addition, confidential virtual machines (VMs), which use AMD SEV [2], or Intel TDX [33], are also actively studied to be integrated with the containers (e.g., Kata Container [36]). Regardless of the underlying technology, current confidential containers assign an enclave for each container, which naturally follows the concept of the container and the enclave.

3 Infeasibility of Confidential Container

However, current confidential containers suffer from large performance overheads [66, 79, 79, 92, 93]. Especially, integrating the enclave with the container imposes i) a longer startup latency due to the creation of the enclave, and ii) a larger memory footprint due to the non-shareable characteristics of the enclave's memory. In order to clearly demonstrate these points, we conducted preliminary experiments as follows.

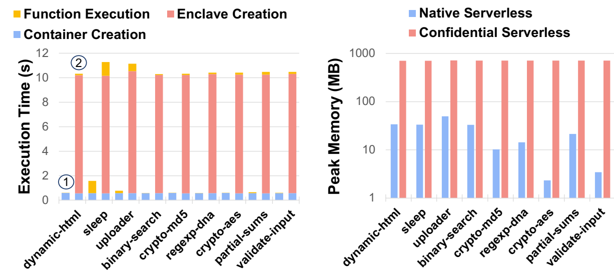
3.1 Preliminary Experiments on the Performance Overheads

We designed the preliminary experiments with two following research questions:

1. How much performance overheads does confidential container impose?
2. Why does confidential container impose such performance overheads?

To this end, we measured the startup latency and memory footprint of two benchmark applications, which are expected to be widely used with confidential containers: i) confidential serverless computing, and ii) confidential database.

Performance Overheads of Confidential Serverless Computing. Serverless computing is an emerging cloud computing model, where the resource management is fully delegated to the cloud providers while the users can solely focus on their workloads [10, 12]. Containers [26, 40] are the key building blocks in such model as they facilitate the resource management and isolation by the cloud providers. In particular, the providers construct a different containerized environment for each request and the following function instance (i.e., unit of the computation in serverless computing [5]), thereby providing a different userland view of resources and resource limits.



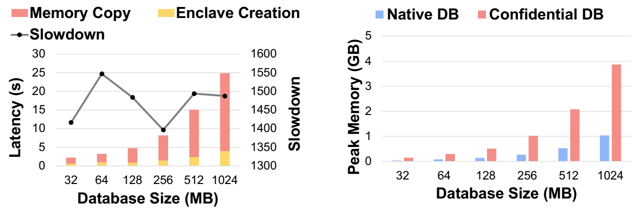
(a) Latency for handling a request. **(b)** Memory usage for a request.

Figure 1. Performance comparison of confidential serverless computing versus the native model. ①: `Native Serverless`, ②: `Confidential Serverless`. The bloated memory of confidential serverless computing is due to the Gramine LibOS's implementation that physically populates all the allocated virtual memory [90].

Confidential serverless computing [79, 92], which employs the confidential containers, is the security enhanced version of serverless computing as it protects the workloads even on a compromised cloud environment. Meanwhile, the providers still manage the system resources so that the users can only focus on their workloads. To this end, state-of-the-art confidential serverless computing frameworks serve each request by creating a new confidential container (including a new containerized process and a new enclave), and running a function instance on it.

Thus, we measured the latency for handling each request and memory footprint of the state-of-the-art confidential serverless computing framework. To be specific, we implemented a security enhanced version of OpenWhisk [5] that runs the functions in the confidential containers, using an SGX enclave [16] and Gramine LibOS [30] as an enclave runtime. As shown in Figure 1, employing current confidential container in serverless computing imposes almost 10× latency slowdown and 20-300× more memory usage. Especially, Figure 1a shows that creating an enclave (for every request) takes 16.8-17.4× longer time than creating a containerized process, demonstrating that creating an enclave (and its security mechanisms) is a major bottleneck.

Performance Overheads of Confidential Database. Database is one of the most widely used applications in cloud computing [47, 49]. In particular, database as a service (DBaaS) [56] is a standard way to quickly integrate the database into the user's service logic while the cloud providers manage the underlying software stacks. DBaaS also heavily relies on the containers as the providers can easily manage and isolate the resources between different service instances.



(a) Latency of forking a child process for snapshot.

(b) Memory usage of database system with fork-based snapshot.

Figure 2. Performance comparison of confidential database system versus the native system.

Especially, the file systems management of the containers facilitates the fork-based snapshots for database systems [50]. In this approach, the parent database process continues to handle the requests by forking a child process, while the child process performs the snapshot by writing the database into the storage. Fork-based snapshot is a widely used approach by database systems [50], as it provides a significant performance benefits in accordance with the copy-on-write semantics of operating systems [23]—i.e., data pages are copied only when a new request to write to that page is received.

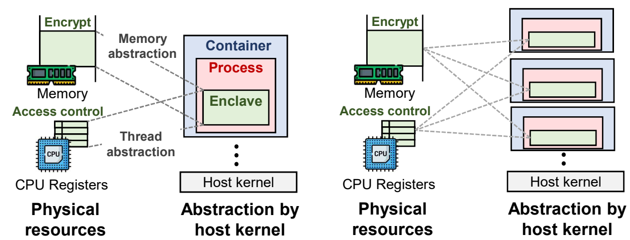
Confidential database, which runs the database system in the confidential containers, protects sensitive data from the cloud providers. With current confidential containers, fork-based snapshot for database would create a new (child) process and enclave, copy the parent enclave's memory into the child's enclave (following the fork's semantics [28]), and run as usual—i.e., the parent serving the requests while the child performing the snapshot. However, copy-on-write semantics would not be allowed as the parent's enclave and the child's enclave cannot share the same protected memory [86].

Thus, we measured the startup latency of fork-based snapshot and memory footprint of current confidential database systems. To be specific, we implemented a security enhanced version of Redis [49] that runs in the confidential container, which employs an SGX enclave [16] and Gramine LibOS [30]. As shown in Figure 2, forking a child process (and copying the enclave) takes over 1000× times longer latency and 4× times larger memory usage as the entire memory contents of the parent enclave need to be copied to the child enclave. While the numbers may be exaggerated by the implementation of Gramine LibOS (i.e., copying memory through TLS encrypted channel [35]), we want to note that inability to share the memory between parent and child enclave is the main cause of these overheads.

Table 1. Previous works to solve the performance issues of enclave.

Research goal	Scheme	Hardware modified
Fast enclave startup	PENGLAI [67]	○
	PIE [79]	○
	LightEnclave [70]	○
	Reusable Enclave [92]	×[†]
Efficient enclave memory sharing	Nested Enclave [83]	○
	PIE [79]	○
	Elasticlave [93]	○
	Cerberus [66]	○

[†]Reusable Enclave discusses only for temporally reusing an enclave.



(a) Previous assumption on enclave. One-to-one enforcement of a process and an enclave.

(b) Key idea of TeeMate. Sharing an enclave across multiple processes.

Figure 3. Relation between an enclave and process in the perspective of host kernel.

3.2 Limitations of Previous Works to Solve the Performance Overheads

Several previous works have tried to solve the performance issues of enclaves as illustrated in Table 1. In particular, we categorize them into two lines of works as follows: i) improving the startup latency of enclave, and ii) enabling the efficient memory sharing between enclaves.

However, most of the works cannot be directly applied to current cloud platforms [13, 14, 20, 32, 46, 52], as they require hardware modification (i.e., shown in Table 1). While Reusable Enclave [92] achieves the goal without modifying the hardware, it enforces the cloud providers to maintain the same container to reuse the enclave, thereby making it difficult to manage the resources. Specifically, all of them have only focused on the inherent issues of enclaves, but not on how to utilize the enclaves in the perspective of host systems.

4 Key Idea of TeeMate

Thus, we re-thought the enclave in the perspective of host systems, and surprisingly, we found an incorrect universal assumption on its usage (§4.1). Based on this observation, we came up with the key idea of TeeMate, which solves the aforementioned issues (§4.2).

4.1 Universal Assumption: One-to-One Enforcement of Process and Enclave

After analyzing the issues, we found there is a universal assumption that a single enclave must be dedicated to only a single process (i.e., one-to-one enforcement of a process and an enclave as shown in Figure 3-(a)). For example of using SGX enclaves, no previous work has assumed using the same enclave by different processes [67, 70, 79, 92], and they implicitly assumed the process which creates the enclave exclusively owns it. While Occlum [86] designs multiprocessing in a single enclave, it is not about how the processes (of the host kernel) use the enclave—i.e., Occlum is also a single process in the perspective of the host kernel.

However, we found that this assumption is wrong. In other words, a single enclave does not have to be dedicated to a single process. This is because an enclave is merely a protected resource composed of an encrypted memory and isolated CPU context, while the process is an abstraction of the resources created by the host kernel—i.e., the way of thinking the resources. In the perspective of host kernel, an enclave can be deemed as any other resources, such as memory and disk, that can be abstracted and shared across the processes.

4.2 Our Solution: Sharing a Single Enclave across Multiple Containers

Based on this observation, we came up with the key idea of TeeMate, sharing a single enclave across multiple containers (i.e., Figure 3-(b)). Since there is no need for an enclave to be dedicated to a single process, it is also possible to share the same enclave across multiple containerized processes. By doing so, we can avoid the performance issues while taking both benefits of the container and the enclave as the conventional confidential containers. In other words, cloud providers can efficiently manage the system resources by applying different containerized environment for each process, while the users can be ensured the security of their data using the enclave. However, we also preserve the performance, as we i) avoid creating a new enclave every time, and ii) enable sharing the enclave's memory between different processes.

To this end, we design TeeMate to provide an abstraction for the host kernel that it is operating different (containerized) processes with dedicated enclaves, but actually using the same enclave. More specifically, we design the primitives to preserve i) the memory abstraction on the shared enclave's memory (i.e., §5.3), and ii) the thread abstraction on the shared enclave's threads (i.e., §5.4). Then, we discuss how the isolation between the processes can be achieved within the same enclave (i.e., §5.5).

5 Design of TeeMate

In this section, we introduce the threat model of TeeMate (§5.1), and explain the design of TeeMate using Intel SGX [16] (§5.2 to §5.5).

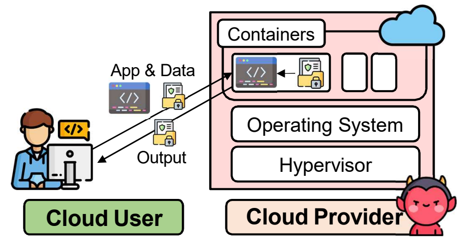


Figure 4. Threat model of TeeMate.

5.1 Threat Model

We assume the common threat model of confidential computing as shown in Figure 4, where the cloud users do not trust the cloud providers. This is because the cloud providers may be compromised, or even themselves are in conflict of interests with the cloud users (e.g., Samsung utilizing the services hosted by Amazon AWS [51]). We want to note that there is a growing demand to protect the data even on a compromised cloud environment as more privacy-senstive data are handled in the cloud.

As explained in §2.2, confidential computing is an emerging solution to meet such needs, so we focus on the performance issues of employing both the confidential computing and container technologies. We do not consider general security issues of confidential computing such as Iago attacks [62] and side channels [75–77, 80]. Denial-of-Service attacks [24, 85] are also out-of-scope. In addition, we trust the implementation of the software components loaded in the enclave, and exploits through their vulnerabilities are out-of-scope. Hardening software implementations is a long been problem, and we believe TeeMate can take advantages of ongoing researches [63, 69, 91].

5.2 Design Overview

As illustrated in §4.2, TeeMate enables high performance confidential containers by sharing the same enclave across different containers (i.e., containerized processes). To this end, TeeMate provide an abstraction for the host kernel that it is operating different processes with dedicated enclaves (within each containerized environment), but actually using the same enclave (inside which, the resources are isolated for each process). Specifically, in order to achieve the process abstractions assumed by the host kernel, TeeMate satisfies two requirements: i) for memory abstraction, sharing the same enclave's physical memory within different virtual address spaces of each process, and ii) for thread abstraction, assigning the threads in the same enclave to each different process.

Thus, we design TeeMate's primitive operations achieving these requirements based on Intel SGX [16]. While we provide the design for Intel SGX only, we want to note the key idea of TeeMate (i.e., sharing the same enclave) is general enough to be applied to confidential VMs (e.g., AMD

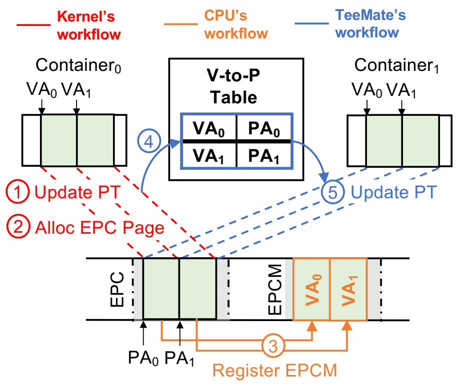


Figure 5. Sharing the same EPC pages across different containers using EPC aliasing

SEV [2], Intel TDX [33]). We discuss how TeeMate can be used for confidential VMs in §9.

5.3 Sharing Enclave Memory across Multiple Containers

In order to provide the memory abstraction, TeeMate maps the physical pages of the same enclave into the virtual address spaces of each process. Thus, TeeMate enables the threads running in different containers (with different address spaces) to access the same enclave's code and data, which avoids to create a new enclave for every new container. Especially in SGX, we name it Enclave Page Cache (EPC) aliasing, as we alias the same EPC pages (of an SGX enclave) to different virtual address spaces—i.e., EPC is a protected memory region used by Intel SGX [16]. In the following, we explain the details how we alias the EPC pages.

Technical Analysis: Address Translation and Validation in SGX. SGX ensures the integrity of the virtual-to-physical address mapping for EPC pages, preventing malicious systems components from launching a page remapping attack (e.g., tricking a victim enclave to access a different EPC page through the same virtual address [65]). This integrity is assured by maintaining additional address mapping within the special EPC page, so called EPCM. Specifically, when a new EPC page is allocated, the kernel updates the page table with virtual-to-physical address mapping (e.g., `VA` to `PA`) related to the EPC page. When the kernel requests the CPU to create the EPC pages, the CPU creates a new EPCM entry per new EPC page, which contains `VA` of the corresponding EPC page located at `PA`. When any access to the EPC page is attempted later using `VA`, the CPU translates `VA` to `PA` using the page table. However, since the page table can be compromised by the adversarial systems components, the CPU further validates that such a translation is correct using EPCM, thereby assuring the integrity of the virtual-to-physical address mapping.

An interesting technical characteristic here is in the address validation mechanism of EPCM, which does not involve the identity of a process (i.e., EPCM does not include process identifiers like ASID [9]). Thus, an EPC page (located at `PA`) can be accessed by any other processes using `VA` as long as the page table of the process contains the same mapping from `VA` to `PA`. If so, the process, which did not initially create the EPC page, can still access the EPC page through accessing to the same `VA`. Note that while this may seem a vulnerable design, it does not harm the security assurance of SGX, which we elaborate the detailed security analysis in §8.

EPC Aliasing. Based on this characteristic, TeeMate aliases EPC pages across different processes. Specifically, EPC aliasing maps multiple virtual pages to the same physical EPC page, where multiple virtual pages are (i) associated with different processes and (ii) these virtual pages have the same virtual address. More technically, if a process p_0 already allocated an EPC page with address mapping (i.e., VA_0 to PA_0), EPC aliasing allows to map the same EPC page to another process p_1 by updating p_1's page table, inserting VA_0 to PA_0 address mapping. After that, the process p_1 can also access the EPC page with virtual address VA_0.

In order to share EPC pages between containers, TeeMate performs the EPC aliasing as illustrated in Figure 5. Specifically, TeeMate creates the initial container. Then, after identifying available virtual and physical pages for an enclave, the kernel updates the page table of the initial container —i.e., adding an address mapping from VA_0 and VA_1 to PA_0 and PA_1, respectively (①). Then, the kernel requests the CPU to allocate new physical EPC pages using the SGX instruction (i.e., `EADD` [65]) (②). This consequently makes CPU to create an EPCM entry corresponding to PA_0 and PA_1, which contains VA_0 and VA_1, respectively (③). At this point, in order to share and thus alias this EPC page later, TeeMate additionally records the mappings for VA_0 and VA_1 (④).

When creating a new container, TeeMate aliases the previously allocated EPC pages. This entails to insert the recorded address mapping—i.e., (VA_0, PA_0) and (VA_1, PA_1)—to the page table of the new container (⑤). Therefore, when the new container accesses the memory with VA_0 or VA_1, the CPU would accordingly translate it to the aliased EPC page located at PA_0 or PA_1 using (i) the page table of the new container and (ii) the corresponding EPCM entry.

From the performance perspective, sharing the enclave's memory significantly reduces the latency since it avoids copying, verifying and encrypting the initial memory contents for every new confidential container. Furthermore, it reduces the memory footprint by enabling to share the common data between two containers, avoiding to duplicate the same data in each enclave every time.

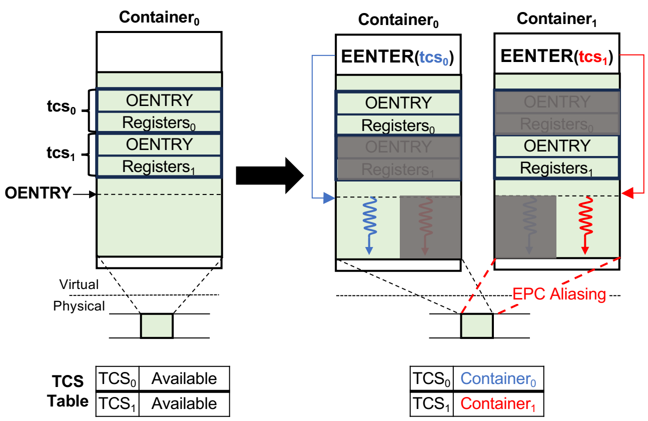


Figure 6. Constructing container-independent enclave threads

5.4 Assigning Enclave Threads to Each Container

Then, TeeMate provides the thread abstraction for the host kernel as if each process is running its own thread in the dedicated enclave. To be specific, TeeMate assigns the threads in the same enclave to each different process so that the host kernel can run each enclave thread in different containerized environments—e.g., providing different file systems using namespace [41]. One may wonder it is a vulnerable design as the host kernel can illegally change the container environment for a given enclave thread, but we can guarantee the security by carefully designing a sanity check logic in the enclave, which is explained in §8. In the following, we elaborate the details how we achieve the thread abstraction in the context of Intel SGX [16].

Technical Analysis: Multi-threads in SGX Enclave. In order to support parallelism, SGX designs unique schemes for secure multi-threading within an enclave, which we refer to as enclave threads [65]. In particular, the execution contexts of enclave threads are managed by Thread Control Structures (TCSs), which are also stored in the EPC pages to protect against adversarial system components. More technically, each TCS manages the entry point and the CPU register context per enclave thread. First, the entry point ensures that an enclave thread always starts or resumes (i.e., `EENTER` or `ERESUME`) at the code address designated in the given TCS (i.e., `OENTRY` field). The entry point often contains security checks and enforcement to sanitize the inputs from non-enclave context. Second, the CPU register context ensures that a paused enclave thread is always resumed as expected. This is carried out by saving (and restoring) all CPU register values to (and from) the TCS[1], which happens when the enclave thread exits (and resumes). It is worth noting that initial TCS pages are measured as other EPC pages, thereby preventing untrusted system components from breaking its initial integrity.

[1]More precisely, TCS stores the reference to State Save Area (SSA) in the EPC pages.

The technical catch here is that unlike the typical use-cases of multi-threading in SGX, we find that an enclave thread does not need to be bound with a specific process. In fact, an enclave thread can be migrated from one process to another. Specifically, when switching from the non-enclave execution context to the enclave execution context, any TCS page can be selected and such a selection does not restrict which process performs the switch (i.e., which process performs `EENTER` or `ERESUME`). TeeMate leverages this operational property to enable individual execution context per container while sharing the EPC memory pages.

Individual Enclave Thread per Container. In order to support an enclave thread per container while aliasing EPC pages, TeeMate designs management schemes to map between a TCS page and a container. As such, while sharing EPC pages, TeeMate assigns a dedicated TCS page per enclave thread, where each enclave thread is associated with a different containerized process.

More technically, Figure 6 shows how TeeMate supports an individual enclave thread per container. In this figure, we assume that a single container (i.e., $\texttt{Container}_0$) and an enclave were initially created before, where no enclave code has been executed yet. Accordingly, TeeMate initializes a TCS table, which indicates that all TCS pages are available. Next, a new container is created (i.e., $\texttt{Container}_1$), which shares the EPC pages through EPC aliasing. To execute enclave threads per container, TeeMate first picks an available TCS page per container (e.g., $\texttt{tcs}_0$ for $\texttt{Container}_0$ and $\texttt{tcs}_1$ for $\texttt{Container}_1$, respectively). Then each container starts the execution of the enclave thread by entering the enclave (i.e., `EENTER`) with the chosen TCS page. Completing the execution of the enclave thread, the TCS table is accordingly updated to mark which TCS page is now returned back to be available.

5.5 Ensuring Isolation Guarantees in an Enclave

The final step toward sharing an enclave is to isolate the enclave's memory between the processes using it (i.e., enclave thread of each process). In case of SGX enclaves, TeeMate cannot rely on page table isolation [38], as untrusted host kernel has the full control of the paging structures. Instead, we implement a software based memory isolation in the enclave as shown in the previous works [57, 70, 86, 88]—i.e., intra-enclave isolation. However, we want to note that it is merely due to the design of Intel SGX [16], and confidential VMs can employ the paging mechanism that is securely implemented by the trusted guest kernel in the VM. We further explain how we implemented the isolation in SGX enclaves for each application in §7.

6 Implementation

TeeMate's implementation consists of i) TeeMate controller, which is a composition of a tailored system software stack (implemented in host kernel) and userspace layer, and

ii) TeeMate runtime, which manages the operations in the enclave.

As TeeMate controller, we first modified Linux SGX driver [43] and Linux kernel [42] for the EPC aliasing (i.e., §5.3). Especially, we implemented new `ioctl` syscalls for remembering the virtual to physical address mappings and populating the same mappings in the other virtual address space. Based on it, we implemented the userspace layer to invoke the `ioctl` for remembering the mappings (by the process that created an enclave), and populating the mappings (by the other process that wants to use the enclave). For the enclave threads, we implemented Linux SGX driver to bookkeep which process uses which TCS page (i.e., §5.4). Then, we made the userspace layer to get the pointer for corresponding TCS page through `ioctl` and enter into the enclave with it. We implemented TeeMate controller with 490 LoC in Linux SGX driver, 10 LoC in Linux kernel, and 250 LoC in Gramine LibOS [30]. It is worth noting that we do not protect TeeMate controller, and its integrity does not affect the security of the components in enclave (i.e., explained more in §8).

For TeeMate runtime which runs in the enclave, we modified Gramine LibOS [30] to support real world applications without modification. To be specific, we reused most of the components in Gramine LibOS, but enabled it to identify the process (based on a requested TCS page) such that the integrity of the files (outside the enclave) can be checked per process. On top of it, we ran actual applications (i.e., Node.js [45] for confidential serverless computing, and Redis [49] for confidential database) while they internally isolate the threads in the enclave. We elaborate more on the intra-enclave isolation for each application in §7. We implemented 1800 LoC of Gramine LibOS for TeeMate runtime, 170 LoC in Node.js for confidential serverless computing, and 270 LoC in Redis for confidential database.

7 Evaluations on Applications

This section evaluates the performance improvement of TeeMate on confidential serverless computing, and confidential database. We first describe the evaluation setup (§7.1), then introduce the evaluation results of each application in the following (§7.2 and §7.3).

7.1 Evaluation Setup

We evaluated TeeMate on 64-core Intel Xeon Gold 6348 CPU machine which supports SGX2 feature [34]. Especially, we ran all the experiments in a QEMU virtual machine [48] with 160GB memory and 48GB EPC size, which runs a Linux kernel 6.2.0 [42].

Models for Comparison. In order to clearly demonstrate the performance improvement of TeeMate, we compare three models (for each application) as summarized below:

Table 2. List of evaluated serverless functions.

Name	Description
dynamic-html	creating html using input
sleep	sleep for 1 second
uploader	upload local file to remote storage
binary-search	binary tree search using input as a key
crypto-aes	AES encryption/decryption
crypto-md5	MD5 hash computation
partial-sums	input array summation
regexp-dna	DNA sequence processing
validate-input	input string processing

- **`Native`** represents the conventional application that does not employ confidential computing (i.e., using only the container). This model shows the maximum performance that TeeMate can achieve.
- **`Strawman`** represents the secured version of the application that employs state-of-the-art confidential container [15, 60]—i.e., each container creates its own SGX enclave. Especially, we used Gramine LibOS [90] as the SGX runtime, and ran the bare-metal application on it.
- TeeMate implements our design such that the same SGX enclave is shared across different containers.

7.2 Confidential Serverless Computing

For the confidential serverless computing, we used OpenWhisk [5], which is a real-world serverless computing platform widely used for analyzing the performance. In particular, for every request, OpenWhisk creates a new container which includes the Node.js [45] runtime, and runs the serverless function implemented using JavaScript (i.e., **`Native`**). Then, the function receives the user's data, runs the code on it, and returns the result. In this respect, we implemented **`Strawman`** model to create a new container, and a new SGX enclave in it, which loads the Node.js runtime [45] (on the Gramine LibOS [90]) to securely run the functions.

Intra-Enclave Isolation for Confidential Serverless Computing. In order to integrate TeeMate with OpenWhisk, we further employed V8 Isolate [54] to isolate the functions in the same enclave. To be specific, for each request, TeeMate's OpenWhisk creates a new container (as usual), aliases an enclave into the new address space, and runs the function using a new enclave thread that is sandboxed by V8 Isolate (i.e., TeeMate). Thus, V8 Isolate ensures the threads cannot access each other's memory, and we can guarantee the containers using the same enclave are isolated (as long as the implementation of V8 Isolate is trusted).

Benchmark Functions. We performed the evaluation with 9 serverless functions written in JavaScript as shown in Table 2. We selected the functions from SeBS [64] and Google Sunspider [29], which cover a wide range of serverless functions that potentially receive sensitive data as input.

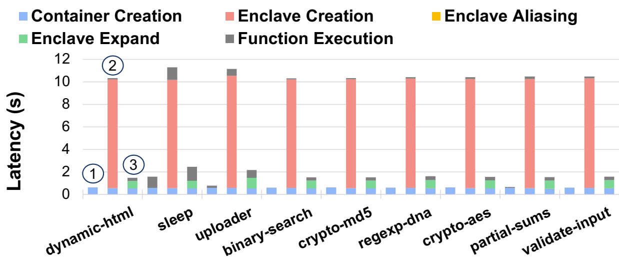


Figure 7. Breakdown of latency for running a function. ①: **`Native`**, ②: **`Strawman`**, ③: TeeMate. "Enclave Creation" includes the time for initializing the enclave and loading software components in it. "Enclave Expand" denotes the time to allocate new EPC pages to accommodate new function instance.

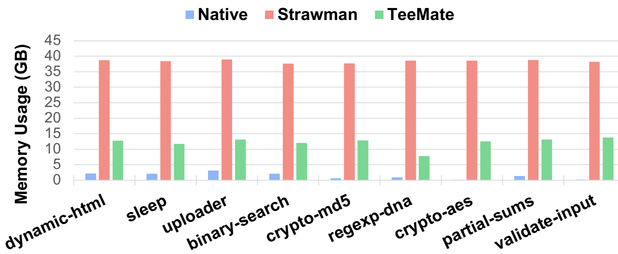


Figure 10. Peak EPC memory usage when 64 concurrent requests are handled.

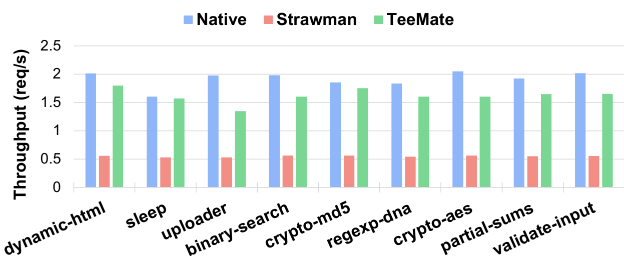


Figure 8. Throughput when 8 concurrent requests are handled.

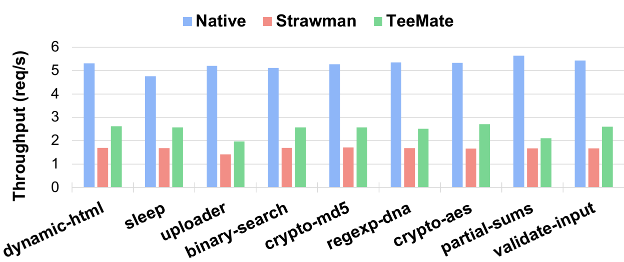


Figure 9. Throughput when 64 concurrent requests are handled.

7.2.1 Performance Improvement of Startup Latency.

Single Latency. We measured the average latency for running each serverless function as shown in Figure 7. TeeMate showed 4.54-6.98× latency speedup over the **`Strawman`** model as it eliminates the overheads to create a new enclave for each function instance (i.e., red portion in the bars). On the other hand, the latency of EPC aliasing (which is added instead of the enclave creation) was significantly low, ranging from 2.32 to 3.01 milliseconds (i.e., less than 1% of the entire latency). **`Strawman`** model consumes almost 10 seconds to initialize an enclave, and load Gramine LibOS and Node.js into it every time (i.e., "Enclave Creation"). However, TeeMate does not impose such overheads, and only needs to allocate new EPC pages for the new function instance, which takes about 27.4-45.6% of the entire latency.

Throughput. We evaluated the throughput improvement of TeeMate by invoking a burst of requests simultaneously and observing the total time to complete all the requests—i.e., throughput is computed as the number of requests divided by the completion time. To this end, we invoked 8 and 64 requests respectively, for each model with each function. As a result, TeeMate exhibited 1.26-3.21× higher throughput than **`Strawman`** as illustrated in Figure 8 (i.e., 8 requests invoked simultaneously), and Figure 9 (i.e., 64 requests). Even when compared to **`Native`** model, TeeMate showed comparable performance by decreasing only 5.5-62% of the native throughput.

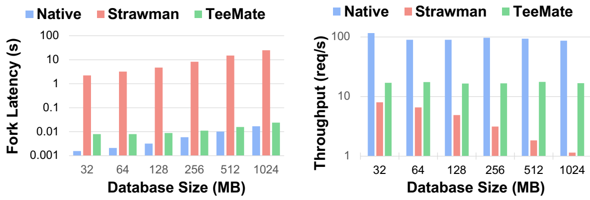
One thing to note is that the throughput gain of TeeMate decreases as the number of requests increases from 8 to 64. We suspect it is because the SGX driver internally reserves a lock [43] when allocating the EPC pages, and the lock contention increases as the number of concurrent requests increases. In order to relieve this lock contention, TeeMate may employ further optimizations such as batch processing the EPC allocation requests [82], or reusing the allocated EPC pages. In addition, we want to note that TeeMate can also create more enclaves for the same function to maximize the throughput, while we used only one enclave for this evaluation.

7.2.2 Performance Improvement of Memory Footprint.

In order to evaluate the memory efficiency of TeeMate, we compared the peak memory usage of **`Native`**, **`Strawman`**, and TeeMate when a bunch of requests are received. For each function, we invoked 64 concurrent requests simultaneously and measured the peak memory usage.

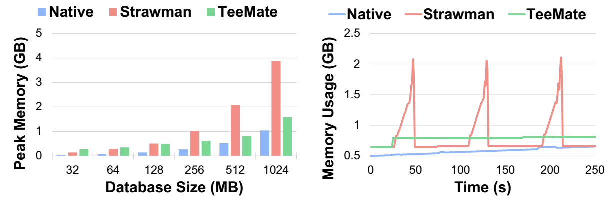
As illustrated in Figure 10, TeeMate showed 2.8-5× lower memory usage compared to **`Strawman`**. This is because **`Strawman`** model needs to load a new runtime (i.e., Gramine library OS [90] and Node.js [45]) within each enclave, while TeeMate needs to create only a lightweight V8 Isolate [54] on top of the shared runtime (in the shared enclave).

On the other hand, TeeMate uses more memory (i.e., 207MB) than **`Strawman`** (i.e., 114MB) when handling only one request. This is because TeeMate needs additional memory to accommodate the isolation mechanisms (i.e., V8 Isolate).



(a) Latency for forking a child. **(b)** Throughput.

Figure 11. Latency evaluation of confidential database.



(a) Peak memory usage **(b)** Overall memory usage

Figure 12. Memory usage of each model

7.3 Confidential Database

For the confidential database, we used Redis [49], which is an in-memory database widely serviced by cloud platforms [74]. Specifically, Redis uses fork-based snapshot [50] to support data persistence. To be specific, Redis process periodically forks a child which performs the snapshot by writing the database into the storage, while the parent continues to handle the requests (i.e., **`Native`**). Copy-on-write semantics [23] are well suited for this mechanism as the full page copy is performed only when a write request is received to the parent.

However, running Redis on state-of-the-art confidential containers cannot employ the copy-on-write semantics as the parent's enclave and child's enclave cannot share the memory (i.e., **`Strawman`**). To be specific, if we run the Redis on Gramine LibOS [90] as usual, fork from the parent creates a new child process (outside the enclave) and creates a new enclave again, then Gramine LibOS would copy all the memory contents from the parent to the child enclave to preserve the semantics of fork—i.e., child process inherits the same memory space as the parent. Thus, the entire pages are copied on fork without supporting copy-on-write.

Intra-Enclave Isolation for Confidential Database. In order to integrate TeeMate with Redis, we implemented software address translation [82] in the Redis, which isolates memory accesses by each process. Especially, after a child process is forked, two threads from the parent and the child still run using the same addresses (in the same enclave with different page tables). However, when a write request is received, we made Redis to copy the target page and execute the request on the copied page (i.e., copy-on-write) such that the other process cannot see the updated contents (i.e., TeeMate). This ensures the memory isolation as the two processes access exactly same physical pages as long as their contents are the same (which does not harm any security guarantee).

7.3.1 Performance Improvement of Fork Latency.

Since the fork-based snapshot necessarily forks a child process, we measured that latency of the Redis process with different size of database. As shown in Figure 11-(a), TeeMate showed 277.6-1046.6× latency speedup compared to **`Strawman`** model. This is because TeeMate does not create a new enclave and copy the parent enclave's memory to the child, but the two processes can share the memory. Fork latency in TeeMate consists of the latency to create a new enclave thread and makes a child process to execute the enclave thread, which is much faster than enclave creation and copying the entire memory. In addition, TeeMate takes the advantages of copy-on-write by which it copies the page only when a write request is received to the parent.

Throughput. We also evaluated the throughput improvement of TeeMate by invoking a burst of requests while the database performs fork-based snapshot. As a result, TeeMate exhibited 2.1-14.6× higher throughput than **`Strawman`** as illustrated in Figure 11b. Since Redis cannot handle any request during the fork system call is handled, the throughput of **`Strawman`** was significantly affected due to the longer fork latency. On the other hand, TeeMate was able to achieve better throughput thanks to the short fork latency and copy-on-write operations.

7.3.2 Performance Improvement of Memory Footprint.

We evaluated the memory efficiency of TeeMate by measuring the peak memory usage and overall memory usage of the Redis while it runs as usual and also performing fork-based snapshots. As illustrated in Figure 12-(a), TeeMate showed much lower peak memory usage compared to **`Strawman`** model as the database size increases. While **`Strawman`** model suffers from large memory footprint due to the duplicated memory of the parent and child enclaves, TeeMate avoids such issues thanks to the shared enclave, showing 41% lower memory usage at maximum.

Meanwhile, when the database size is small, TeeMate showed slightly larger memory usage because it needs additional memory for software address translation. In addition, we observed the memory overhead incurred by the heap allocator of Gramine LibOS when creating a new thread [30].

We also evaluated the overall memory usage of each model as described in Figure 12-(b). For this evaluation, we set the database size to be 512MB and generated random requests to the database engine. Additionally, we set the database engine to create a snapshot once every minute, which is the same as default configuration of Redis [50]. As expected, TeeMate showed significantly lower memory usage than

`Strawman` model when performing fork-based snapshot operations. However, in normal situations when not performing a snapshot, TeeMate showed slightly higher memory usage due to the aforementioned reasons.

8 Security Analysis

In this section, we analyze the security guarantees of TeeMate.

Security Guarantees of Sharing an Enclave across Multiple Processes. While it seems unsafe at first glance to share a single enclave across multiple processes, it does not compromise the security guarantees of confidential computing [17]. The key question is that "what if the compromised host enters the enclave in the context of another process (different from the one that initially created the enclave)?". This is the same question as "what if the compromised host modifies all the states of a process (except the memory of the enclave) to be the other one, and enters the enclave?". The second question is already the common threat model of confidential computing, and widely discussed in the academia [2, 8, 16, 33]. The answer is that "the host cannot compromise the enclave as all the security critical data and logic should be located in the enclave"—i.e., the security guarantees of the confidential computing (and those of TeeMate) still hold.

The key is that the code in an enclave should not believe any information passed over from the outside enclave. In other words, all the security critical data should be managed in the enclave, which includes the identity of the client who invoked the request, TLS encryption key, and the hash of the files to be checked for integrity. All the security operations such as decrypting the ciphertext from users, or checking the integrity of opened file should be performed in the enclave also, which is the same as the conventional use-cases [90]. For example, if a compromised host runs an enclave thread in a different containerized environment (that was not supposed to be used), then, access to a security critical, but different data should be detected in the enclave.

Security Limitations of TeeMate. Compared to the previous approaches that protect the workloads of each process using each dedicated enclave [67, 70, 79, 92], TeeMate has weaker isolation guarantees as it employs intra-enclave isolation. The intra-enclave isolation should be implemented on software (e.g., V8 Isolate [54]) in case of Intel SGX, while it can be implemented using the paging mechanism in case of confidential VMs (e.g., AMD SEV [2], and Intel TDX [33]). However, we believe TeeMate still achieves the major goal of confidential computing—i.e., removing cloud providers from the trusted path. Furthermore, we want to note that software fault isolation (that TeeMate should use in case of Intel SGX) is already widely used in various real world scenarios (e.g., WASM sandbox in browser [55], kernel model sandboxing [81]). While these may have vulnerability, TeeMate can benefit from ongoing researches to improve the software fault isolation.

9 Discussion

9.1 Applicability to VM based Confidential Computing

The core observation of TeeMate is that an enclave does not have to be bound to a process as the enclave is just a set of physical resources while the process is the abstraction managed by host kernel. This does not depend on the type of enclave, and it is also applied to the VM based confidential computing (e.g., AMD SEV [2], and Intel TDX [33]). This is because the VM is also a set of physical resources (i.e., virtual memory and virtual CPU). While Linux kernel manages a single VM by a single process [37], the VM can be shared as long as the memory and thread abstractions are preserved.

In this respect, we demonstrate that design primitives of TeeMate can be applied to AMD SEV-SNP VMs. Specifically, we check that multiple processes can share the same enclave pages (i.e., secure pages in AMD terminology) of a single SEV-SNP VM. In the following, we provide a brief explanation of access control and address translation mechanism of SEV-SNP, and then describe how we leverage these mechanisms to adopt the design of TeeMate in SEV-SNP.

Technical Analysis of SEV-SNP: Access Control and Address Validation Mechanism. ASID (Address Space Identifier) is a crucial element of access control mechanism in AMD SEV-SNP [3]. The ASID serves as a unique identifier for each VM, and it is used to select a VM encryption key (VEK). VEK is used for encrypting and decrypting enclave pages. Thus, only the VM with the correct ASID can access the enclave pages encrypted with the corresponding VEK. Importantly, the ASID can be modified at runtime by the hypervisor during a VMEXIT event [78].

SEV-SNP employs a reverse map table to validate the address translation when a VM accesses an enclave page. When a VM allocates new enclave page, the reverse map table records the VM's ASID and the guest physical address of the page. When the VM later accesses the enclave page, the hardware checks whether the VM's ASID and the guest physical address match the corresponding record in the reverse map table. This prevents malicious hypervisor from launching a page remapping attack, similar to purpose of address translation mechanism in SGX (mentioned in §5.3).

Enclave Page Aliasing in SEV-SNP. Based on above analysis, we propose a method to alias enclave pages in SEV-SNP by matching two key pieces of information: i) ASID, and ii) guest physical address. To share the same enclave pages (i.e., secure pages of the same VM) between multiple processes, the hypervisor should first record the ASID of the VM which allocates the pages. Then, when another process attempts to access the page, the hypervisor should assign the ASID of the VM that originally allocated the page. Furthermore, the hypervisor needs to assign the same nested page table to ensure that the enclave page is accessed using the same guest physical address.

We verify through experiments that our method actually works in SEV-SNP machine. We modified Linux KVM module to change ASID and nested page table pointer (nCR3) of VM during VMEXIT event to those of the VM that originally allocated the page. We will also investigate the applicability of TeeMate to other confidential VM technologies such as Intel TDX [33], and ARM CCA [8] in the future work.

9.2 Use-cases of TeeMate

TeeMate enables multiple processes (that are managed by untrusted host kernel) to efficiently share the data within an enclave. Thus, this feature can be widely applied to the confidential computing use-cases that need frequent data communications, while the resources are managed by untrusted host kernel (e.g., micro-service architecture using Kubernetes [39], or big data analysis using Spark [6]). However, it should be carefully applied as this approach necessarily bloats the trusted computing base (TCB) by implementing the control logic in the enclave.

10 Related work

Confidential Container. Confidential container is gaining popularity due to its ability to meet the needs of efficient resource management by cloud providers and data protection by cloud users using TEE (Trusted Execution Environment). SCONE [60] is one of the first confidential container system, which integrates SGX enclave [16] to Docker container [26]. Additionally, the Cloud Native Computing Foundation's Confidential Container project [15] is actively conducting various research related to confidential containers. For example, the project implements both process-level containers using Intel SGX [19] and microVM-level containers [18] using confidential virtual machines like Intel TDX [33] and AMD SEV [2]. TZ-Container [72] utilizes ARM TrustZone to create a secure execution environment for each container process. Although many confidential container system have been proposed, they have not challenged to the universal misconception that only one process can use a specific enclave.

Confidential Serverless Computing. Incorporating TEE with serverless computing is gaining more interests as it provides strong security guarantees to protect sensitive data and code even in a compromised environment. Clemmys [89] uses SGX enclave to block platform provider from introspecting the memory, and devised a cryptographic model to prevent maliciously modifying the order of function chaining from the platform provider. AccTEE [68] and S-FaaS [59] introduced a fair and trustworthy resource accounting for confidential serverless computing. SEVeriFast [71] implemented new bootstrap scheme in AMD SEV [2] for low startup latency in VM-based confidential serverless computing. Reusable enclave [92] achieved low startup latency in confidential serverless computing by secure enclave reset mechanism. PIE [79] extends the SGX design through hardware modification to optimize the startup latency and function chaining latency by memory sharing between enclaves. Several works have also provided the ground for sharing memory between the enclaves [66, 93], but all of them need to modify the hardware, and are not able to be applied to current platforms. It is worth noting that the memory sharing approaches mentioned above focus on sharing memory between different enclaves, whereas TeeMate focuses on sharing a single enclave across multiple containers.

Confidential Database. Confidential database protects database engine on untrusted cloud so that the confidentiality and integrity of data and queries are guaranteed. For instance, EnclaveDB [84] used SGX enclave to protect all database state including the data and query from the cloud provider. Library OS for Intel SGX such as Graphene-SGX [90] and Haven [61] are also used to run confidential database without application modification. OBLIVIATE [58] proposed data oblivious filesystem to prevent side-channel attacks against malicious cloud provider toward database engine.

Intra-Enclave Isolation. For the performance and security issues, several previous works have suggested to split the enclave into multiple isolated regions. Chancel [57] protects the client's data in application, which handles each client's request using dedicated thread. Chancel uses a compiler-based per-thread isolation inside enclave. Occlum [86] enables multi-process applications to run efficiently inside enclave by software fault isolated processes, leveraging Intel MPX and code instrumentation. These isolation approaches can also be used for the isolation mechanism of TeeMate, but we want to note that V8 Isolate is more appropriate for isolating high-level programming languages that are widely used in serverless computing (e.g., JavaScript, and Python). LightEnclave [70] and EnclaveDom [88] leveraged Intel MPK for fine-grained isolation inside enclave. However, it needs hardware modification under the threat model of confidential computing as Intel MPK basically depends on host kernel.

11 Conclusion

This paper proposes TeeMate, which introduces a new approach to utilize the enclaves in the host's perspective. Especially, we found there is a universal misconception that an enclave must be dedicated to a single process that created it, and we break this assumption by sharing an enclave across multiple processes. To this end, we design the primitives to preserve the memory and thread abstraction for a single SGX enclave to be shared across multiple processes. Based on it, we implemented confidential serverless computing and confidential database, and demonstrated that TeeMate shows significant latency speedup and memory usage reduction compared to the same applications using state-of-the-art confidential containers.

References

[1] Amazon ECS. https://aws.amazon.com/ecs/?nc1=h_ls. Accessed: 2023-06-07.

[2] AMD Memory Encryption. https://www.amd.com/system/files/TechDocs/memory-encryption-white-paper.pdf. Accessed: 2023-06-07.

[3] AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf. Accessed: 2023-10-30.

[4] Amdsev github. https://github.com/AMDESE/AMDSEV. Accessed: 2024-05-22.

[5] Apache OpenWhisk: Open Source Serverless Cloud Platform. https://openwhisk.apache.org/. Accessed: 2023-06-07.

[6] Apache Spark. https://github.com/apache/spark. Accessed: 2023-11-30.

[7] Arm Confidential Compute Architecture. https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture. Accessed: 2023-06-07.

[8] ARM Confidential Compute Architecture (ARM CCA). https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture. Accessed: 2023-11-17.

[9] Arm cortex-a series programmer's guide for armv7-a. https://developer.arm.com/documentation/den0013/d/The-Memory-Management-Unit/Multi-tasking-and-OS-usage-of-translation-tables/Address-Space-ID. Accessed: 2023-11-30.

[10] AWS Lambda. https://aws.amazon.com/lambda/?nc1=h_ls. Accessed: 2023-06-07.

[11] Azure confidential computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/. Accessed: 2023-06-07.

[12] Azure Functions. https://azure.microsoft.com/en-us/products/functions/. Accessed: 2023-06-07.

[13] Azure software-as-a-service. https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-saas/. Accessed: 2024-05-22.

[14] Cloud4c kubernetes-as-a-service. https://www.cloud4c.com/kr/solutions/kubernetes-services. Accessed: 2024-05-22.

[15] CNCF Confidential Container. https://www.cncf.io/projects/confidential-containers/. Accessed: 2024-05-22.

[16] Computing with Intel SGX. https://gcore.com/cloud/intel-sgx. Accessed: 2023-06-07.

[17] Confidential computing. https://en.wikipedia.org/wiki/Confidential_computing. Accessed: 2024-05-22.

[18] confidential-containers : confidential-containers. https://github.com/confidential-containers/confidential-containers. Accessed: 2024-05-22.

[19] confidential-containers : enclave-cc. https://github.com/confidential-containers/enclave-cc. Accessed: 2024-05-22.

[20] Containers in Google Cloud. https://cloud.google.com/containers?hl=en. Accessed: 2024-05-22.

[21] Containers vs VMs. https://www.redhat.com/en/topics/containers/containers-vs-vms. Accessed: 2023-11-07.

[22] Control Groups. https://docs.kernel.org/admin-guide/cgroup-v1/cgroups.html. Accessed: 2023-06-07.

[23] Copy-on-write. https://en.wikipedia.org/wiki/Copy-on-write. Accessed: 2024-05-22.

[24] Denial-of-service_attack. https://en.wikipedia.org/wiki/Denial-of-service_attack. Accessed: 2024-05-22.

[25] Digital Sovereignty at AWS. https://aws.amazon.com/compliance/europe-digital-sovereignty/. Accessed: 2024-05-22.

[26] Docker: Accelerated, Containerized Application Development. https://www.docker.com/. Accessed: 2023-06-07.

[27] Enhance the user experience through advanced security features for confidential computing. https://www.cloudsigma.com/intel-confidential-computing/. Accessed: 2023-06-07.

[28] fork (system call). https://en.wikipedia.org/wiki/Fork_(system_call). Accessed: 2024-05-22.

[29] Google Sunspider. https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/tsx-programming-considerations-ov.html. Accessed: 2023-10-27.

[30] Gramine Library OS with Intel SGX Support. https://github.com/gramineproject/gramine. Accessed: 2024-05-22.

[31] Grand View Research - Market Size & Trends. https://www.grandviewresearch.com/industry-analysis/saas-market-report. Accessed: 2024-05-22.

[32] IBM Cloud Functions. https://cloud.ibm.com/functions/. Accessed: 2023-11-06.

[33] Intel Trusted Domain Extensions (Intel TDX). https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html. Accessed: 2023-11-17.

[34] Intel xeon gold processor. https://www.intel.com/content/www/us/en/products/details/processors/xeon/scalable/gold.html. Accessed: 2023-11-30.

[35] IT Security Guidelines for Transport Layer Security v2.1. https://english.ncsc.nl/binaries/ncsc-en/documenten/publications/2021/january/19/it-security-guidelines-for-transport-layer-security-2.1/IT+Security+Guidelines+for+Transport+Layer+Security+v2.1.pdf. Accessed: 2023-11-30.

[36] katacontainers. https://katacontainers.io/. Accessed: 2023-10-30.

[37] Kernel-based Virtual Machine. https://www.ibm.com/docs/en/license-metric-tool?topic=connections-kernel-based-virtual-machine. Accessed: 2024-05-22.

[38] Kernel page-table isolation. https://en.wikipedia.org/wiki/Kernel_page-table_isolation. Accessed: 2024-05-22.

[39] Kubernates: Production-Grade Container Orchestration. https://kubernetes.io/. Accessed: 2023-06-07.

[40] Linux Container. https://www.redhat.com/en/topics/containers/whats-a-linux-container. Accessed: 2024-05-22.

[41] Linux Kernel Namespaces. https://www.kernel.org/doc/html/latest/admin-guide/namespaces/index.html. Accessed: 2023-06-07.

[42] Linux Kernel v6.2. https://github.com/torvalds/linux/tree/v6.2. Accessed: 2023-06-07.

[43] linux-sgx-driver. https://github.com/intel/linux-sgx-driver. Accessed: 2023-06-07.

[44] Microservice Architecture. https://www.redhat.com/en/topics/microservices/what-are-microservices. Accessed: 2024-05-22.

[45] Node.js. https://nodejs.org/en. Accessed: 2023-06-07.

[46] Oracle cloud infrastructure (oci). https://www.oracle.com/cloud/. Accessed: 2024-05-22.

[47] PostgreSQL Database Management System. https://github.com/postgres/postgres. Accessed: 2024-05-22.

[48] QEMU, A generic and open source mahcine emulator and virtualizer. https://www.qemu.org/. Accessed: 2023-06-07.

[49] Redis: In-memory Database. https://github.com/redis/redis. Accessed: 2024-05-22.

[50] Redis Persistence. https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/. Accessed: 2024-05-22.

[51] Samsung Migrates 1.1 Billion Users across Three Continents from Oracle to Amazon Aurora with AWS Database Migration Service. hhttps://aws.amazon.com/solutions/case-studies/samsung-migrates-off-oracle-to-amazon-aurora/?nc1=h_ls. Accessed: 2024-05-22.

[52] Software-as-a-service on aws. https://aws.amazon.com/saas/?nc1=h_ls. Accessed: 2024-05-22.
[53] statista: Kubernetes - statistics & facts. https://www.statista.com/topics/8409/kubernetes/#topicOverview. Accessed: 2024-05-22.
[54] V8 Isolate Class Reference. https://v8docs.nodesource.com/node-18.2/d5/dda/classv8_1_1_isolate.html#ac58181f2e183f10173d480881b5d42a2. Accessed: 2023-06-07.
[55] Webassembly Security. https://webassembly.org/docs/security/. Accessed: 2024-05-22.
[56] What is DBaaS? https://www.oracle.com/database/what-is-a-cloud-database/dbaas/. Accessed: 2024-05-22.
[57] Adil Ahmad, Juhee Kim, Jaebaek Seo, Insik Shin, Pedro Fonseca, and Byoungyoung Lee. Chancel: Efficient multi-client isolation under adversarial programs. In *NDSS*, 2021.
[58] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. Obliviate: A data oblivious filesystem for intel sgx. In *NDSS*, 2018.
[59] Fritz Alder, N. Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. S-faas: Trustworthy and accountable function-as-a-service using intel sgx. In *Cloud Computing Security Workshop (CCSW)*, 2019.
[60] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Josuha Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L Stillwell, David Goltzsche, Dave Eyers, Rudiger Kapitza, Peter Pietzuch, and Christof Fetzer. Scone: Secure linux containers with intel sgx. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.
[61] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.
[62] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.
[63] Tobias Cloosters, Johannes Willbold, Thorsten Holz, and Lucas Davi. {SGXFuzz}: Efficiently synthesizing nested structures for {SGX} enclave fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3147–3164, 2022.
[64] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: a serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference (Middleware)*, 2021.
[65] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.
[66] Lee Dayeol, Kevin Cheang, Alexander Thomas, Catherine Lu, Pranav Gaddamadugu, Anjo Vahldiek-Oberwagner, Mona Vij, Dawn Song, Sanjit A. Seshia, and Krste Asanovic. Cerberus: A formal approach to secure and efficient enclave memory sharing. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2022.
[67] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the {PENGLAI} enclave. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.
[68] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. Acctee: A webassembly-based two-way sandbox for trusted resource accounting. In *Proceedings of the 20th International Middleware Conference*, pages 123–135, 2019.
[69] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities. In *NDSS*, 2023.
[70] Jinyu Gu, Bojun Zhu, Mingyu Li, Yubin Xia, and Haibo Chen. A hardware-software co-design for efficient intra-enclave isolation. In *USENIX Security Symposium*, 2022.
[71] Benjamin Holmes, Jason Waterman, and Dan Williams. Severifast: Minimizing the root of trust for fast startup of sev microvms. In *Proceedings of the 29th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 1045–1060, 2024.
[72] Zhichao Hua, Yang Yu, Jinyu Gu, Yubin Xia, Haibo Chen, and Binyu Zang. Tz-container: Protecting container from untrusted os with arm trustzone. In *Science China Information Sciences*, pages 1869–1919, 2021.
[73] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–32, 2018.
[74] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. Shieldstore: Shileded in-memory key-value storage with sgx. In *Proceedings of the Fourteenth European Conference on Computer Systems*, 2019.
[75] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. Dawg: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 974–987. IEEE, 2018.
[76] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020.
[77] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Netcat: Practical cache attacks from the network. In *IEEE Symposium on Security and Privacy*, 2020.
[78] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. Crossline: Breaking "security-by-crash" based memory isolation in amd sev. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.
[79] Mingyu Li, Yubin Xia, and Haibo Chen. Confidential serverless made efficient with plug-in enclaves. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 306–318. IEEE, 2021.
[80] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.
[81] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. Preventing kernel hacks with hakc. In *Proceedings of the 2022 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2020.
[82] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 238–253, 2017.
[83] Joongun Park, Naegyeong Kang, Taehoon Kim, Youngjin Kwon, and Jaehyuk Huh. Nested enclave: Supporting fine-grained hierarchical isolation with sgx. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 776–789. IEEE, 2020.
[84] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *IEEE Symposium on Security and Privacy*, 2018.
[85] Christoph L Schuba, Ivan V Krsul, Markus G Kuhn, Eugene H Spafford, Aurobindo Sundaram, and Diego Zamboni. Analysis of a denial of service attack on tcp. In *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No. 97CB36097)*, pages 208–223. IEEE, 1997.
[86] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single enclave of intel sgx. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 955–970, 2020.

[87] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *NDSS*, 2017.

[88] Marcela S.Melara, Michael J. Freedman, and Mic Bowman. Enclavedom: Privilege separation for large-tcb applications in trusted execution environments. *arXiv preprint arXiv:1907.13245*, 2019.

[89] Bohdan Trach, Oleksii Oleksenko, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. Clemmys: Towards secure remote execution in faas. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 44–54, 2019.

[90] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *USENIX Annual Technical Conference*, pages 645–658, 2017.

[91] Yuanpeng Wang, Ziqi Zhang, Ningyu He, Zhineng Zhong, Shengjian Guo, Qinkun Bao, Ding Li, Yao Guo, and Xiangqun Chen. Symgx: Detecting cross-boundary pointer vulnerabilities of sgx applications via static symbolic execution. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2710–2724, 2023.

[92] Shixuan Zhao, Pinshen Xu, Guoxing Chen, Mengya Zhang, Yinqian Zhang, and Zhiqiang Lin. Reusable enclaves for confidential serverless computing. In *USENIX Security Symposium*, 2023.

[93] Jason Zhijingcheng Yu, Shweta Shinde, and Prateek Saxena. Elasticlave: An efficient memory model for enclaves. In *USENIX Security Symposium*, 2022.