Lightweight, Secure and Stateful Serverless Computing with PSL

Alex Thomas
Shubham Mishra
Eric Chen
John D. Kubiatowicz

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-191

http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-191.html

October 25, 2024

Lightweight, Secure and Stateful Serverless Computing with PSL

Alexander Thomas
alexthomas@berkeley.edu
UC Berkeley
USA

Shubham Mishra
shubham_mishra@berkeley.edu
UC Berkeley
USA

Kaiyuan Chen
kych@berkeley.edu
UC Berkeley
USA

John Kubiatowicz
kubitron@berkeley.edu
UC Berkeley
USA

Abstract

We present PSL, a lightweight, secure and stateful Function-as-a-Serivce (FaaS) framework for Trusted Execution Environments (TEEs). The framework provides rich programming language support on heterogeneous TEE hardware for statically compiled binaries and/or WebAssembly (WASM) bytecodes, with a familiar Key-Value Store (KVS) interface to secure, performant, network embedded storage. It achieves near-native execution speeds by utilizing the dynamic memory mapping capabilities of Intel SGX2 to create an in-enclave WASM runtime with Just-In-Time (JIT) compilation. PSL is designed to efficiently operate within a asynchronous environment with a distributed tamper-proof confidential storage system, assuming minority failures. The system exchanges eventually consistent state updates across nodes while utilizing release-consistent locking mechanisms to enhance transactional capabilities. The execution of PSL is up to 3.7x faster than the state-of-the-art SGX WASM runtime. PSL reaches 95k ops/s with YCSB 100% read workload and 89k ops/s with 50% read/write workload. We demonstrate the scalability and adaptivity of PSL through a case study of secure and distributed training of deep neural network.

ACM Reference Format:
Alexander Thomas, Shubham Mishra, Kaiyuan Chen, and John Kubiatowicz. 2024. Lightweight, Secure and Stateful Serverless Computing with PSL . In . ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Trusted Execution Environments (TEEs) are becoming a popular way to deploy security-critical applications in untrusted environments. TEEs, such as Intel Software Guard Extensions (SGX) and AMD Secure Encrypted Virtualization (SEV), execute privacy-preserving applications in *secure enclaves* with confidentiality, integrity, and strong isolation from the untrusted kernel or infrastructure. Unfortunately, developing and managing applications on TEEs can be error-prone and complex. Inadvertent programming errors can result in long development cycles and private data leakage.

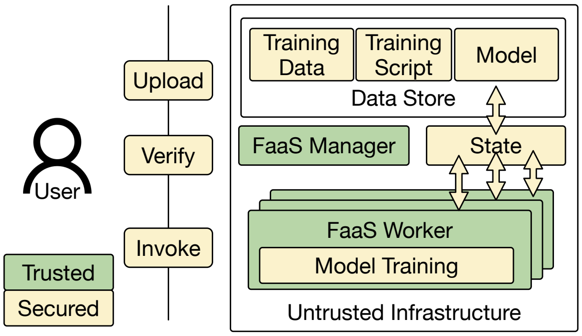


Figure 1. Use Case of PSL in Distributed Learning Training on Confidential Data. In PSL, we consider privacy-preserving distributed deep learning training on untrusted infrastructure. Contrary to the conventional secure enclave programming interface that forces users to protect their data with hand-crafted remote encryption schemes, PSL provides Function-as-a-Service (Faas) with simple and easy-to-use interfaces, in which users exploit standardized mechanisms to *store* the function and parameters to servers, *verify* that an expected function will be executed with confidentiality and integrity, and *invoke* the function.

In contrast, serverless computing, with the Function-as-a-Service (FaaS) model, is an emerging computing paradigm that streamlines application development and deployment without the complexity of building and maintaining the infrastructure. With FaaS, one can decompose data-intensive applications into many functions and exploit massive parallelism and scalability of cloud infrastructure.

In this paper, we show how FaaS becomes an effective option for TEE-based applications by offloading the burden of managing underlying cryptographic hardware infrastructure and state management to the framework. Our secure FaaS framework scales easily across different computing infrastructures, such as multi-cloud and edge-cloud environments, with varying security policies.

Statefulness, a property provided by some of the FaaS frameworks, such as Faasm [40] and Cloudburst [42], allows applications to get access to a Key-Value Store (KVS) to exchange the states with other FaaS workers. Instead of relying on some external database service, stateful FaaS enables state sharing natively, thereby increasing system availability and lowering the latency of propagating state updates. Statefulness is useful for data-intensive applications, such as video processing [48], machine learning [8], and robotics [25, 26]. However, the *way* in which statefulness is integrated into a FaaS environment is a crucial consideration for in-enclave applications to reduce the risk of inadvertent data leakage.

By way of example, Figure 1 shows a distributed learning algorithm that trains on private data. A stateful and secure FaaS framework can streamline the development and deployment of this application by shielding the user from having to manage the enclave environments of multiple workers, verify that workers are running correct code, protect and secure training data through encryption and signatures, and schedule communication for updates to model data.

Challenges in stateful FaaS have been greatly discussed and mostly addressed in out-of-enclave setting, such as [40, 42], and challenges in providing a usable execution framework have been proposed such as Graphene [43], Occlum [38]. However, limited systems have been proposed in in-enclave stateful FaaS due to the following challenges:

Execution Model: Heterogeneous TEE hardware involves varied programming capabilities, leading to a trade-off among *language support, portability*, and the capability to *isolate the application from the runtime.*

For instance, AMD SEV provides secure enclaves as confidential VMs, thus programming akin to standard cloud virtual machines. In contrast, Intel SGX[1] , one of the dominating secure enclaves available on Intel processors, requires that predefined functions calls from untrusted operating system to enclave and from enclave to untrusted operating system. While this reduces the Trusted Computing Base (TCB) by only including required components, it necessitates specialized designs and modifications to both the program and its associated libraries.

In the basic SGX model, the binary is required to be statically linked to ensure the loaded code pages cannot be modified, leading to large binaries and limited modularity and portability. The basic model also does not provide confidentiality of the executable – a significant limitation in today's untrusted environments.

Providing an efficient execution model that unifies heterogeneous TEE hardware is important. Existing work uses interpretation: for example, S-FaaS [4] only supports interpreted runtimes like Javascript and WebAssembly Micro Runtime (WAMR) or use ahead of time (AOT) compilation that requires compiling to the correct target CPU and special care needed to work inside secure TEEs.

State Security: Since TEEs do not guarantee secure or trusted storage, state out of the enclaves needs to be confidential and tamper-proof. This requires encryption and signatures on the critical path of propagating state updates. Naively applying conventional encryption and signature schemes leads to an inefficient protocol with compute-bound bottlenecks. Any system using such cryptographic scheme should also have a dedicated key management infrastructure.

Consistency: Existing TEE-based consistent state management generally use consensus protocols for total order over all operations being executed in the system. We argue that this is not the perfect fit for FaaS. Firstly, not all applications need a total order. Using a leader-based consensus protocol like Raft [33] (as used in CCF [24]), leads to lower overall throughput as only the leader node is allowed to propose writes. However, the strong total order and consensus might limit the scalability of FaaS workers. Secondly, the churn in a FaaS environment is supposed to be much higher than a distributed database setting. Consensus protocols generally have a reconfiguration phase which causes a node joining the system to have some added startup latency. This latency is critical for FaaS applications that are generally short-lived.

To resolve these challenges, we observe the need for a TEE-based stateful FaaS framework that: (1) supports multiple languages and TEEs with low overhead, while also protecting the confidentiality of the application itself; (2) isolates runtime from the application workers; (3) supports efficient confidential state sharing with options for locking, but only when necessary; (4) guarantees durable eventually consistent state updates.

To this end, we present PSL, a secure, lightweight and stateful FaaS framework. PSL has the following properties, summarizing our contributions:

- **PSL supports a portable binary with native support for multiple programming languages and heterogeneous enclave hardware**. It uses create an in-enclave WebAssembly (WASM) runtime to provide multi-language support at near-native speed. It provides support for popular secure hardware backends (Intel SGXv2 and AMD SEV) which can be extended to other secure hardware given a list of supported primitives.
- **PSL enables Just-In-Time compilation that outperforms state-of-the-art SGX WASM runtimes.** Unlike traditional use of SGX, our system also permits statically compiled binaries to be encrypted for privacy and unpacked only within the protections of an enclave. We use the dynamic memory mapping capabilities of Intel SGX2 with just-in-time compilation.
- **PSL enables secure state persistence in untrusted storage.** Data outside the enclaves is always encrypted,

[1] SGX is available for all Intel processors before 2022, and currently available for all Intel Xeon servers. This work assumes its latest generation (SGXv2).

hashed, and signed, and durably stored in a majority quorum of storage servers.

- **PSL supports scalable, efficient, and eventually-consistent state updates with release-consistent locking.** We provide a state update protocol that works in an asynchronous network environment and performs well even though updates are signed, hashed, and encrypted for secure storage in the network. With 8 FaaS workers, it provides a throughput of 89k ops/s with a 50-50 read-write YCSB workload with all cryptographic protocols enabled.

2 Background and Motivation

We introduce the background of secure enclave execution and discuss the limitations of existing approaches in current secure and stateful FaaS frameworks.

2.1 Execution Models in TEE

Trusted Execution Environments (TEEs) create *enclaves* which are secure, isolated environments protected from the privileged host OS, hypervisor, and any hardware devices connected to the host. There are at least two different classes of TEEs in wide use.

The first is the *confidential VMs* as provided by AMD SEV and the upcoming Intel TDX. These systems provide a programming model akin to standard cloud virtual machines with extra protection. Confidential VMs typically involve a large trusted computing base (TCB), where the user needs to trust all the binaries and drivers in the VM image provided by the infrastructure provider. The challenge in this environment is to *reduce* the TCB as much as possible.

The second is exemplified by Intel SGX, which provides a more restricted programming model that provides a secure execution container more akin to a process and that requires predefined function calls from the untrusted operating system to enclave (ecalls) and from enclave to untrusted operating system (ocalls). While this structure reduces the TCB by only including required components, it necessitates specialized designs and modifications to both the program and its associated libraries.

Providing an execution model for confidential VMs is relatively straightforward, since any VM image that works on typical cloud machines should work with minimal changes and performance degradation in a Confidential VM domain. As a result, we focus on providing an execution model that works for Intel SGX. We then extend our SGX-driven execution model to Confidential VM environments by providing a restricted unikernel-based runtime with minimal functionality to mirror the needs of our SGX environment.

Existing execution models in Intel SGX can be typically categorized as following:

1. *Static linking*: Conventional SGX usage statically links all the dependencies within a single binary. SGX requires loading all code pages at startup time for code integrity, which incautious design may end up loading with gigabytes of the binary. Conventional SGX usage also prevents binaries from being encrypted, thereby exposing algorithms to external analysis.
2. *Interpreter*: Some frameworks, such as S-FaaS, statically link a language interpreter such as Javascript. This approach provides limited language support, and cannot support more complex interpreted languages such as python.
3. *Library OS (LibOS)*: Existing work implements minimal Library OS that typically proxies the system calls to the host system out of the secure enclave. With Library OS, one can use as Linux with only one process. Although existing container orchestration frameworks (*i.e.*, Kubernetes), can be utilized here, the container must be reloaded to switch to another application.
4. *Sandboxed Runtime*: Using a sandboxed runtime has become a more popular option, given the rise of WebAssembly. Recent systems compile WebAssembly MicroRuntime (WAMR) to SGX enclaves to dynamically interpret and execute WebAssembly.

2.2 In-Enclave Runtime

The Function-as-a-Service (FaaS) model in cloud computing enables users to access the cloud infrastructure without configuring it. Consequently, we believe that FaaS is an attractive option for users wanting the security of TEEs without the hardship of configuration. Conventional use of SGX by separating trusted and untrusted components of code and statically linking with libraries that are compatible with a target TEE is at odds with FaaS; thus, we automatically rule out static linking of this type. Although LibOS can allow for running unmodified applications, it does not provide an execution environment. While an interpreter is generally an attractive option for FaaS, it provides lackluster performance compared to running native code. We focus on WebAssembly with its rich language support and maturity of different runtimes that advertise near to native speeds.

2.3 A Case for a FaaS JIT Runtime

WebAssembly supports three execution modes: (1) runtime interpretation that directly interprets bytecode without first compiling to native machine code, (2) Ahead-Of-Time (AOT) that compiles WASM bytecode into native machine code, and (3) Just-In-Time (JIT) that compiles WASM bytecode to native code at runtime. Runtime interpretation can overcome the static linking requirement posed by SGX, because interpretation avoids the overhead associated with compiling code and the code is executed within the interpreter itself; however, the cost of decoding instructions on the fly is a significant overhead. AOT compiles WASM to native

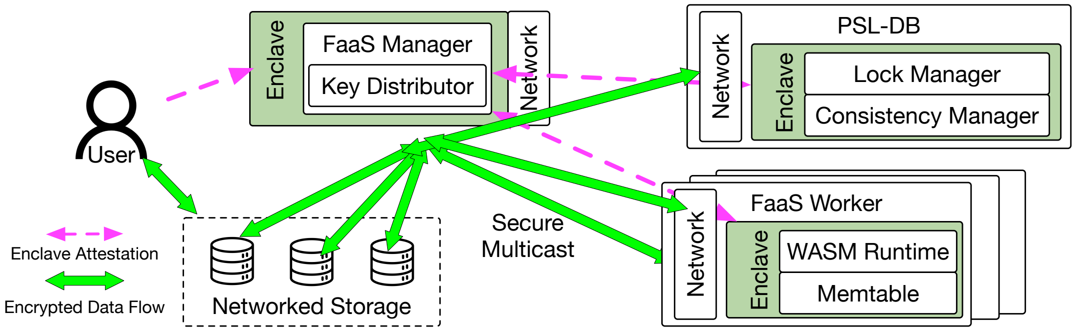


Figure 2. The architecture of PSL: The FaaS Manager launches and attests all worker enclaves. Enclave workers cache key-value pairs in their local Memtable, and share state in an eventually consistent way using a Secure Concurrency Layer (SCL) that takes advantage of a secure-multicast primitive within the network. The PSL-DB tracks the most recent version of the state of each key and pushes updates to a set of replicated networked storage servers that provide state persistence.

code that is ready to execute as soon as it is loaded with a similar mechanism as Intel SGX's static loading but requires knowing the right flags to configure and the correct libraries to link that are compatible within an enclave.

In contrast, JIT allows users to compile a single binary that can be run on any JIT-enabled runtime. The JIT runtime can automatically link the correct dependencies and compile the code to become compatible with its secure hardware backend. Furthermore, JIT's compilation process can utilize dynamic information, potentially optimizing the code based on runtime conditions and the specific hardware it is running on. There is also a security advantage when using a JIT runtime, as using JIT ensures that malicious or unknowing clients with a faulty compiler cannot break the WebAssembly sandbox. Compiling things inside a TEE adds further assurance that the sandbox doesn't become compromised. There has been work done on verifying the sandboxing of WASM binaries [28], but it currently is for nightly builds and is not suitable for a low latency FaaS environment. With all that said, JIT doesn't come for free, as the drawback to using JIT is poor cold start-up time, which is critical in a FaaS environment. We attempt to optimize start-up time latencies to make it more practical in a FaaS environment.

2.4 State Management

While secure enclaves protect the data during computation, special care must be taken to protect the confidentiality and integrity of data at rest and in transit. Scaling up with multiple FaaS eventually requires sharing of state among the workers. This raises interesting consistency questions and query performance issues as blocks of data as a whole must be exchanged or stored encrypted and, therefore, can't be queried at a finer granularity.

Management of state in FaaS generally takes one of three forms: (1) Stateless: The workers use an external database service to store and retrieve state. (2) Centralized: The workers use an in-cluster database to coordinate state updates among themselves. (3) Decentralized and eventually consistent: The workers periodically exchange their local cache with each other and merge their updates using a Conflict-free Replicated Datatype (CRDT). For use in enclave-based FaaS, the decentralized approach is a better fit due to its higher resilience to failures and attacks.

Another line of work runs State Machine Replication (SMR) within enclave-based systems. While this is a similar approach to the decentralized case above, it provides a stronger total order guarantee, which may not be necessary for many applications.

3 Overview

We assume a heterogeneous compute environment with varied security policies and hardware configurations. For example, components of PSL could be spread over multiple geo-distributed cloud regions, even going across multiple cloud providers. Another example of a target heterogeneous environment is FaaS workers running in an edge environment where the data is persistently stored in the cloud.

3.1 Threat and Network Model

PSL adopts the typical cloud attackers who can listen and tamper with any communications or computations. For example, the attack may come from a compromised operating system kernel or a malicious staff member, both situations in which the attacker has full control over the operating system. PSL guarantees the confidentiality, integrity, and provenance of any data in execution and in transit. The trusted computation base (TCB) of PSL is limited to the processor chip, codebase, and the WASM runtime running in an enclave, which explicitly excludes the operating system managed by the cloud provider. PSL does not guarantee against side-channel attacks, given that Intel SGX suffers from various side-channel vulnerabilities [14, 16, 39]. Various techniques [14, 32, 39, 41] proposed to mitigate the risk of side-channel attacks for enclaves.

Since any entity outside the TEE can be malicious, we developed our consistency protocol to work in an asynchronous and Byzantine environment. A network adversary can arbitrarily drop, delay, or replay packets. Workers in our system can be put in network partitions for a finite but arbitrarily long time. Our consistency protocol does not rely on timeouts for any action (e.g., view changes in consensus terminology) as a malicious OS can forever hold the system's progress in a livelock state.

We also assume the existence of a collision-resistant hash function (e.g., SHA256), a secure digital signature (e.g., Ed25519), and an authenticated encryption scheme (e.g., AES-GCM)

3.2 System Overview

Figure 2 shows the components of PSL. The FaaS Manager is responsible for launching FaaS workers on users' requests. It has Key Distributor running inside an enclave which is responsible for securely distributing user's keys and inputs to the FaaS workers using attested TLS channels. The FaaS worker contains a WASM Runtime, which runs JIT-compiled WebAssembly functions submitted by the user, with the securely sent inputs. Multiple FaaS workers store their recent state updates in an in-enclave buffer called Memtable. We build a **Secure Concurrency Layer (SCL)** that encrypts and signs the Since by assumption, messages can be lost in the network, we have a special long-running FaaS worker, the PSL-DB, which makes sure every worker sees linearizable updates in state.

For durability, we assume the existence of **replicated network-embedded storage servers**. Particularly, to tolerate f independent failures, we require $n = 2f + 1$ storage servers. We always store to a majority quorum. Persistent data is always encrypted using the user's application key. The user also stores the function binary and the inputs and retrieves the output from these replicated storage servers.

3.3 Key Management

Every PSL user generates two keys: (1) *Application Encryption Key*, (2) *Application Signing Key*.

After attestation (see Figure 3), the Key Distributor establishes TLS channels between it and the FaaS workers and the user. The user sends these two keys through the TLS channel which in turn is sent to the FaaS workers. The storage servers only get the public key corresponding to the Signing key which they use to verify the signature on each block they store.

If the signature algorithm supports, the Key Distributor can generate child keys using a Hierarchical Deterministic Wallet approach [23] and send one child key to each worker. This allows fine-grained control over key access. The storage servers are given the parent public key which they can use to verify signatures from any child key.

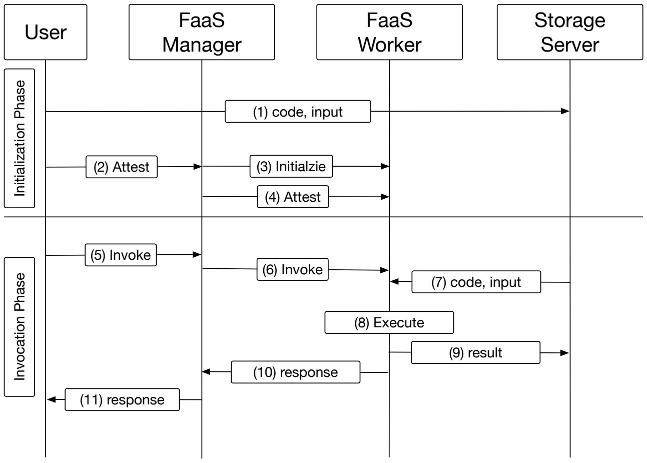


Figure 3. FaaS Manager Launch Sequence Diagram: At initialization time, (1) user securely uploads the function and input to storage server. (2-4) user attests FaaS manager while FaaS manager initializes and attests FaaS workers. When user invokes function, (5) they send a request to FaaS manager and (6) FaaS Manager distributes the request to other workers. The workers pull from storage server and execute the function. (9-11) On completion, the worker uploads the result to storage server and the FaaS Manager routes the response back to the user. The detailed description of the Launch Sequence Diagram and the security protocol can be found in Section 4.1 with the same step indexing.

4 PSL Executor

4.1 Transitive Secure Worker Initialization

Conventional secure enclave requires users to directly attest the remote machine and establish a secure connection between the user and the remote machine. This prevents man-in-the-middle attack. However, FaaS is a different compute paradigm that the workers should be pre-initialized before users issue any request. Then users can enjoy fast launching time and call the functions executed on remote machines without concern for infrastructure setup. As a result, we introduce a FaaS manager, an indirection proxy between workers and user, that facilitates the FaaS initialization process.

Figure 3 shows the lifecycle of an application in PSL:

(1) At initialization phase, user uploads the encrypted function code and function input to storage server with a key of hashed uploaded data. The code and function are encrypted with a symmetric key private to the client
(2) The client attests FaaS Manager with standard enclave attestation protocol to establish a secure TLS channel
(3) the client sends symmetric key to FaaS Manager through the secure channel
(4) FaaS Manager initializes and boots FaaS workers and their runtime environment. This step corresponds to typical

FaaS frameworks which initiate a pool of active workers that stand by and await for requests.
(5) FaaS Manager attests FaaS workers and establish a secure TLS channel to each worker
(6) At run time, client invokes the function with the keys from storage server of the function and input
(7) FaaS manager finds idle workers, and send with the keys from storage server and the symmetric encryption key
(8) FaaS worker retrieves the input and function from storage server
(9) FaaS workers execute the function based on the input and exchange the intermediate results with other workers.
(10) At the end of the execution, FaaS worker stores the result in storage server and returns FaaS manager with the corresponding key.
(11) FaaS Manager return client with the key. Client retrieves from storage server and gets the final result.

Sandbox Reloading. Loading and attesting each client function per invocation for an enclave would have an unreasonable initialization time. As a potential optimization, we argue for reusing the execution environment across worker functions. In this case, we save round trip time due to attestation and bootloading time for the enclave. How do we ensure that reusing execution environments across workers is safe? We propose using a secure sandbox to disallow client functions from hijacking the worker enclave. The FaaS manager is responsible for launching and pre-attesting worker enclaves to ensure that they run the correct sandbox. Once the worker enclave finishes execution, it maintains a memory array that was allocated and used throughout the worker's execution. The sandbox will force the worker to only write to memory within this memory array, which is freed and zero'd out after the worker finishes executing or aborts due to errors.

4.2 WASM JIT Runtime

In section 2.3, we argued for using a JIT runtime engine as our FaaS worker execution model. Implementing a secure JIT runtime engine is paramount. SGX1 did not allow for dynamic memory mapping, so in order to even allow a JIT runtime engine in SGX1, you needed to allocate a sufficiently large, static, and executable reserved section, which would be where the generated code resides. First, with the heteregenous workloads and memory consumption of FaaS, it isn't sufficient to have a static memory reservation for all workers. Second, as the developers cannot call mprotect on the reserved section, they risk having a code injection attack that might hijack the sandbox.

Thus, our solution for our runtime engine should satisfy the following properties 1) the amount of memory that the runtime engine allocates for the worker should be dynamic 2) and the runtime engine should have the ability to unmap or change the permissions of pages dynamically. Enclave dynamic memory mapping (EDMM) support was introduced in SGX2, which enables mapping and modifying the permission of pages dynamically after the enclave has been measured. We use EDMM to dynamically allocate memory for each client as well as unmap and clear the client's memory when execution is completed.

We port WAVM [37], a WASM JIT runtime engine that relies on the LLVM framework for code generation, inside of SGX. We modify WAVM to use EDMM to mmap buffers in-enclave for the client. The WAVM runtime has a loader which fetches code keyed on a hash outside the enclave. Once the code is fetched, the runtime will verify that the code is correct by hashing the module and matching it with the original hash. Once the code is verified, it is mapped and loaded, then goes to the LLVM pipeline. This involves emitting LLVM intermediate representation (IR), optimizing the IR, then generating the machine code. The memtable is protected and managed by the runtime, so that potentially malicious clients are sandboxed from accessing memtable state that it doesn't own.

The JIT runtime also links in our own WASM module that interfaces with the KVS with simple ReadKey/WriteKey interfaces. Clients simply need to include a header file with no other dependencies. On top of the KVS interface, we create shared memory through familiar array abstractions that we call PSLArray. Multiple workers can easily share data by writing to a PSLArray, where updates are transparently multicasted to other workers. Later, we demonstrate how one can use a PSLArray to have a shared distributed weight array that multiple workers read and write to for deep learning training.

5 Secure Concurrency Layer (SCL)

In this section, we formally define our consistency guarantees and system constraints with our system design.

To the application running in a FaaS worker, our state management mechanism interfaces as a shared memory key-value store. We define three key guarantees of our system: (1) Monotonicity, (2) Eventual Progress, (3) Validity. We define these formally below:

Definition 1. ***Monotonicity.*** *If a worker reads value V_1 for some key k, all subsequent reads for the same key k from the same worker returns values V_2 such that*

$$V_1 \leq_e V_2$$

for some partial order $\leq_e$ on the space of values.

Definition 2. ***Eventual Progress.*** *If a worker writes value V_1 for some key k, every worker in the system will eventually read values V_2 for the key k, such that,*

$$V_1 \leq_e V_2$$

Note that, we operate in an asynchronous environment where packets can be delayed or dropped arbitrarily. Eventual progress guarantees that updates made by a worker are visible to all other workers eventually. However, it does not provide time bound for the progress to be visible.

Definition 3. ***Validity.*** *If a worker reads value V for some key k, there exists a worker that wrote the same value V for the key k.*

This property guarantees that no network adversary can maliciously inject state updates into the system.

We observe the following constraints in a TEE-based FaaS system: Firstly, TEE platforms often come with limited available memory. For example, Intel SGX has an Enclave Page Cache (EPC) size of only 128MB. The encrypted paging becomes the performance bottleneck for large applications. Hence, we must be efficient in the in-memory cache usage of our application. Note that this is different from the current design of systems like CCF [36], which assumes that the application state is completely in memory. Secondly, we do not assume that the TEE-enabled machines have large disk capacities. We make this distinction due to our heterogeneous infrastructure assumption. Thus, all access to persistent data must be through network-attached storage servers. Thirdly, for confidentiality, all data sent outside an enclave should remain encrypted. This incurs additional costs of encryption on the critical path of sending messages. Also, queries on data cannot be performed outside the TEE.

Existing systems gravitate towards running a consensus protocol among the FaaS workers. This makes these systems perform active replication: every worker first agrees on a common total order of commands or transactions and then executes them to update their local state. We argue that, if the application does not need a strict total ordering of its commands, active replication under-utilizes the guarantees a TEE provides. A trusted environment guards against malicious code execution. If the application logic (provided by users of our system) is correct, multiple FaaS workers could be running multiple transactions at the same time and merge the results eventually. This model of *passive replication* is more suited for a TEE-based environment.

5.1 Merge Operation

With the above constraints in mind, we now discuss our system design for eventual consistency. Central to this discussion is the partial order $\leq_e$ used for merging two values for the same key. We assume each value V is structured as $V = (v, ts)$ where v is the actual data and ts is a timestamp attached to it. We define the partial order as follows:

$$V_1 \leq_e V_2 \iff (ts_1 < ts_2) \lor (ts_1 = ts_2 \land H(v_1) \leq H(v_2))$$

where $H(.)$ is a collision-resistant hash function.

Many eventually consistent systems [18] use vector clocks for their timestamps to capture causality, but the size of vector clocks is linear to the number of workers. Due to memory limitations in our system, we did not use a vector clock, rather, we used a Lamport clock. This works because the only communication between our FaaS workers is through this eventually consistent key-value store system. When timestamps are equal, we use the hash of data to break the tie instead of node identities. This mitigates the security problem that a network adversary may influence the result of the merge by selectively dropping packets from workers.

5.2 Durable Writes

Every FaaS worker has a local cache of the writes in the system, called the **Memtable**. To the application worker, we expose a transactional interface that publishes a batch of writes at a time. We only guarantee durability on commit. For every key-value update, the Memtable increases the Lamport timestamp by one.

Figure 4 describes the operations that take place on Commit. We assume the existence of $n = 2f + 1$ storage servers, where at most f can fail. The replication on this set of storage servers is controlled by the FaaS worker itself.

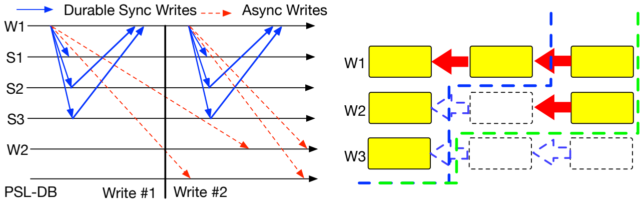
In the FaaS worker, whenever the application performs a write, the writes are all stored in a transaction buffer. Once the application calls Commit on the transaction, these writes are first stored in the Memtable and then multicast to every other worker in the system. Simultaneously, these writes are sent to all the storage servers and the worker waits for $f + 1$ of them to return an acknowledgment. The commit completes once the acknowledgment arrives. We attach the hash of the previous multicast block to the current block of writes and give it an increasing sequence number. Then the block is encrypted using an authenticated encryption scheme before sending it out. The storage servers only see the encrypted blocks, which are also periodically signed by the FaaS worker.

Once other workers receive a multicast block, it applies all the writes to its own Memtable. If the keys already exist, it uses the partial order $\leq_e$ to merge the values. Other workers can miss some writes due to packet drops, but that does not break the monotonicity guarantee.

5.3 PSL-DB Consistency Manager

PSL-DB periodically generates global snapshots of all writes made by all the FaaS workers. It facilitates workers to recover missed messages and acts as a database for keys not present in the workers' Memtables. It needs to be alive for the entirety of the application's lifetime.

The PSL-DB receives the multicast messages from workers and keeps track of their sequence numbers to detect missing blocks. If a missing block is detected, the PSL-DB uses hash pointers attached to the later block it received to back-fill the missing block from storage servers. It also periodically



(a) Sequence Diagram For Durable Commits

(b) View of PSL-DB with Consistent Cut

Figure 4. (a) **Durable Commits.** Worker W_1 multicasts a batch of write to storage servers (S_1, S_2, S_3), other workers (W_2) and PSL-DB. It only waits for f + 1 responses from storage servers and progresses to multicast the next batch. (b) **View of PSL-DB.** It has all the writes from W_1. It lags behind W_3 and it has missed an intermediate write from W_2. The blue line shows a causally consistent cut. The green line is not a causally consistent cut until the PSL-DB back-fills the missing block from W_2.

requests all storage servers to return the most recent block sent by each worker and waits for the response from f + 1 storage servers. The block returned with the maximum sequence number is guaranteed to be the most recent block from the worker. If this sequence number is higher than the one the PSL-DB has seen, it uses this block and its attached hash pointer to back-fill any block that it might have missed.

This ensures that the PSL-DB's view is always a causally **consistent cut**[10] of all writes by each worker (i.e., the subset of all writes seen by PSL-DB at any point is such that for all write blocks in the set, its attached hash pointer points to another write in the set, or is null). Due to message drops, it may be lagging behind the most recent writes.

The PSL-DB has its own Memtable which is used to merge all key-value pairs in its received blocks. It periodically flushes the Memtable out into a Checkpoint block which is also stored in a quorum of storage servers. Then it compresses the Checkpoint into a **Sync Report** which contains the list of keys and their corresponding timestamps and hash of the value stored in the checkpoint and the most recent sequence number seen from each worker. This Sync Report is then multicast to all the workers. The workers use the Sync Report to query the keys that are more updated in PSL-DB than their own Memtable. The Sync Report also contains an increasing sequence number and a hash pointer to the previous Sync Report and is replicated in $f + 1$ storage servers. So, a worker can also recover and back-fill missed Sync Reports from storage servers using the hash pointers. PSL-DB can also answer queries for keys, either returning directly from its Memtable, or sending the hash of the Checkpoint block that has the most recent value for the key.

5.4 Release-consistent Locking

Some applications require additional mutually exclusive reads and writes using locks. PSL-DB implements *release-consisent locking* [21] with a Lock Manager that grants locks on worker's requests. Once a worker releases a lock, it attaches the hash of its most recent write block with a lock release message to the Lock Manager. The PSL-DB, on receiving the message, makes sure it has seen that block and then issues a Sync Report. The Sync Report is also attached with the lock grant message to the next worker which requests to acquire the lock, thereby making sure that that worker is updated with all the writes made by the previous writers. This makes the locking system release consistent and allows the application to have total order, albeit at the cost of performance.

5.5 Discussion

Memtable size. To avoid thrashing and improve application performance, the size of a Memtable should fit the working set of keys. We provide a low cache mode if writes from other workers can evict a key in the cache before being included in a Sync Report: (1) Pins keys not included in a Sync Report to the cache. Any attempt to evict these keys from the Memtable blocks until a Sync Report with this key included arrives. (2) Does not let multicast blocks from other workers introduce new keys in the Memtable. This preserves linearizability even when a recently written key is forced to be evicted due to low Memtabl size.

Garbage Collection. Note that, the blocks made durable in storage servers by the workers serve as a Write-Ahead Log (WAL) for the PSL-DB. Once the PSL-DB puts the writes of these blocks into a Checkpoint, all future reads happen from this Checkpoint block only. Hence, once a block is added to a Checkpoint, it can be garbage-collected from the storage servers.

Behavior under failures. We assume that the FaaS workers have a crash-stop failure model. We don't allow a restarted FaaS worker to reuse its old identity. Note that, PSL-DB is *not* a single point of failure in the system. Although we do all our evaluations with a single PSL-DB, multiple PSL-DBs could be active at any given point in time, without a need for coordination among them. In that case, the workers need to send their key-value pair queries to the PSL-DB from which they received the most recent Sync Report. A PSL-DB can crash mid-operation and be restarted directly. The back-filling strategy triggered by multicast from the workers will make sure that before the next Sync Report, the PSL-DB has seen all the blocks from all the workers. While the PSL-DB is in a crashed state, the workers still keep making progress using the multicast mechanism. However, since we consider an asynchronous network environment, all queries to the PSL-DB for key-value pairs and all lock requests will be blocked until the PSL-DB comes back up and catches up with all the workers.

5.6 Correctness

We state the following lemmas:

Lemma 5.1. *Monotonicity with eventual progress imply linearizability.*

Lemma 5.2. *Under an asynchronous network environment, the protocol described above provides linearizability, eventual progress, and validity.*

We omit the proofs here for brevity.

6 Implementation

We implement PSL with ∼ 21k lines of C/C++. We use a fork of the OpenEnclave framework for our Intel SGX-specific code that supports EDMM. In order to port WAVM, we ported the LLVM libraries to link with OpenEnclave's framework. We create point-to-point persistent TCP channels using ZeroMQ. Communication among FaaS workers, the PSL-DB and storage servers happens using an RPC format. We use Ring Buffers [31, 46] to asynchronously send messages to enclave threads. This circumvents the need for Ecalls for every message. The implementation of our FaaS worker can scale up to use each core available in a machine effectively. We use one server thread to receive messages and two for processing app launch, key exchange, multicast, and Sync Report messages. The rest of the cores can be used to run the application. The Memtables for FaaS workers are implemented using LRU caches.

The storage servers are implemented using RocksDB for persistence. We disable write-ahead logging and use an 8GiB write buffer size for performance reasons. We also provide two alternative designs, one directly using the underlying file system (e.g., ext4), and another providing a shim layer over a cloud storage service (e.g., Azure Blob Storage). We noticed that for blocks smaller than 4MiB, RocksDB provides stable and high write throughput, as the blocks are mostly cached in memory. For larger blocks, the file system design is better since it frequently fills the write buffer in and causes unnecessary compactions in RocksDB. Since most of our experiments generate blocks smaller than 4MiB, we use the RocksDB system for our evaluation. The cloud storage version of the storage server provides high-latency operation.

For authenticated encryption, we use AES-GCM with 256 bit keys. AES-GCM generates a small tag containing the Galois MAC of the data encrypted. This tag is sent along with the encrypted data itself. Anyone holding the symmetric decryption key can verify the integrity of the encrypted message using this tag. We use RSA with 2048 bit key size for key exchange with the FaaS manager. For all digital signatures, we use Ed25519. We previously used seckp256k1. Whereas secp256k1 provided more fine-grained control over key management by using a child key derivation scheme like Hierarchical Deterministic Wallet [1], Ed25519 has an order of magnitude better performance in signing and verification. We use SHA256 as our collision-resistant hash function. All crypto operations are programmed in OpenSSL3.0.

We mention a few notable optimizations below.

6.1 Runtime Optimizations

Reducing Startup Costs. In order to reduce start-up latency, we first identify the the phases that account for the bulk of the cold start up latency 1) allocating memory for the worker and 2) generating machine code. We predict and pre-allocate memory to hide the latency required during cold start up time for each worker. To prevent machine code generation, we cache modules generated by JIT. We further allow users to AOT compile binaries to save on cold start latency costs. We demonstrate how much speed-up we achieve through these optimizations in our evaluation section.

Concurrency. We enable a promise-based concurrency model in our system. The main application thread can run any function asynchronously and get a promise, which it can use to wait for the result later on. If there is only one application thread available, promises behave as if they are run synchronously. However, if more than one application thread is available, the promises are handed to idle threads for parallel execution.

6.2 SCL optimizations

Multicast Batching. Since the Memtable is shared among application threads and the multicast thread, updating the Memtable with updates from multicast messages requires acquiring a lock for mutual exclusion. To reduce lock contention, it is preferable to have the update time short. After every iteration of the multicast updates, we check if there are more than one multicast updates in the Ring Buffer. If so, we collect all of them together and batch them into one big multicast block. In the next iteration of updates, we use this big block for updates. This eliminates redundant keys in the original batch. Each key only appears once in the big block.

PSL-DB LSM tree. Since PSL-DB is also used as a database for key-value pairs, requesting a key from PSL-DB should be made fast. However, PSL-DB also runs in a limited memory enclave and has network-attached storage servers only. Under these design considerations, we adapt the Log-structured Merge (LSM) tree approach taken by databases like RocksDB and LevelDB. We implement the Memtable using C++ maps. When the Memtable is filled with a threshold number of keys, we flush the Memtable and cache the resulting checkpoint block in memory. A fixed-length list of cached checkpoint blocks makes the level-1 in LSM tree terminology. The level-2 consists of a sorted set of keys. The keys are divided into ranges, where the PSL-DB has to keep the first key of each range in its memory. The rest of the range is durably stored in storage servers and is also held in another cache. When level-1 fills up, a constant fraction of the blocks are compacted into level-2 and then erased from memory. This is where the design differs from traditional LSM trees. Whereas generally, LSM trees fetch data from disk to perform compaction, we maximally utilize the in-memory cache by performing

Operation	Median latency (μs)	99.99^{th} percentile latency (μs)
ICMP Ping	641.5	1337.7
Writing to quorum	1000	180000
AES-GCM on 2KiB block	2.0	39.1
Ed25519 Sign	44.0	61.0
Ed25519 Verify	150.0	745.3
RocksDB write	7.0	357.2

Table 1. PSL Cluster System Characteristics

the compaction eagerly in memory. The sizes of all these levels can be configured. For fetching keys, we first search the Memtable, then the level-1 blocks in descending order, and then finally level-2. The keys returned from level-1 and level-2 do not contain the actual values. Rather, they contain the hash of the Checkpoint block that has the value. This is done to prevent double buffering. The FaaS worker requesting the key should, for locality, fetch the Checkpoint block and apply all its writes to its Memtable.

7 Evaluation

7.1 Evaluation Setup

We use Azure DCds_v3 series machines with Intel(R) Xeon(R) Platinum 8370C CPU @ 2.80GHz and Linux kernel 6.5 for our Intel SGX2 environments. For our experiments on SCL, we use a cluster of 10 machines with 8 vCPUs and 64GiB memory. For our experiments on the runtime, we use 2 machines with 16 vCPUs and 128GiB memory. For storage servers, we use three Azure D8d_v5 machines with 8 vCPUs and 32GiB memory. We use a partition of 128GiB ephemeral storage. We set up a Kubernetes cluster using K3S with the SGX2 and storage machines where we set the resource requests in such a way that each entity (FaaS manager, PSL-DB or worker) gets a machine by itself.

7.2 Execution Microbenchmarks

In Figure 5, we compare with another SGX runtime Twine [30], which uses AOT compilation and is based off of the Web Assembly MicroRuntime (WAMR). We run the Polybench [35] suite which is a collection of compute bound benchmarks compiled using Emscripten [2] with -O3. For Twine, the benchmarks were compiled using WAMR's own AOT compiler (wamrc) with -O3. We also compare PSL against running the benchmarks on WAVM outside of SGX. PSL outperforms Twine in a majority of the benchmarks, even achieving up to 3.7x speedup over Twine. In order to ensure that the performance is not simply due to the differences in runtime, we normalize the performance relative to running WAVM outside of an enclave and compare against WAMR, the runtime Twine uses. We see that the majority of benchmarks WAVM and WAMR perform around the same, yet PSL's performance over Twine remains strong.

Compared to native performance, we expect slowdown as previous work has shown that running in WASM generates more memory operations, produce more instructions that branch, and generates more instructions [27]. A cache miss in SGX is costly, relative to outside an enclave, as the memory has to be decrypted when it goes into the processor. For example, the correlation benchmark of Polybench does operations on > 20 MiB, which is the size of the L1 + L2 cache combined. This accounts for why 2 of the benchmarks, correlation and covariance have a significant slowdown due to L1/L2 cache missing. Note that the same slowdown also appears in Twine.

7.3 Startup Latency

In Figure 6, we show the performance of an active worker retrieving a invocation request up until it returns the first instruction. Cold start up latency for JIT scales with the binary code size as generating machine code dominates cold start latency. As expected, for cold AOT, the start up doesn't scale as much as JIT with binary size. For both JIT and AOT, caching the generated code significantly improves performance. For PSL workers, this demonstrates the importance of pre-fetching and warming up the cache with libraries we expect the user to link with.

7.4 SCL Benchmarks

Workloads. Unless stated otherwise, we use the YCSB[15] benchmark to test SCL. We use a workload of 300,000 unique keys with 100 byte values. Since we are operating in a FaaS environment, we run YCSB from trace files, rather than using a client machine. Workload trace files with varying read-write ratios are generated per worker and stored in the storage servers. The workers are given the hash of the trace files as input. The workers retrieve its file from storage servers and start processing requests in batches. We use a batch size of 20 in our experiments. We let each experiment run for at least 5 minutes and sample throughput at every 5s interval once it is stabilized.

Scaling. We run the YCSB workload with 0, 50, and 100% write ratio. Figure 7 shows the scaling of the system for 1, 2, 4, and 8 FaaS workers running 6 application threads each. We achieve up to 95k operations per second for the 100% read case. The 100% write case does not scale as well due to multicast traffic.

Cryptographic overhead. From Table 1, we know that the individual cryptographic operations take 10s μs. We run an eight worker setup with number of 1, 3, and 6 application threads and vary the amount of signatures done. Figure 8 shows the aggregate throughput achieved by the system. The overhead of signatures is $\sim$ 10%. The system saturates the network as well as the CPU cores in the machine.

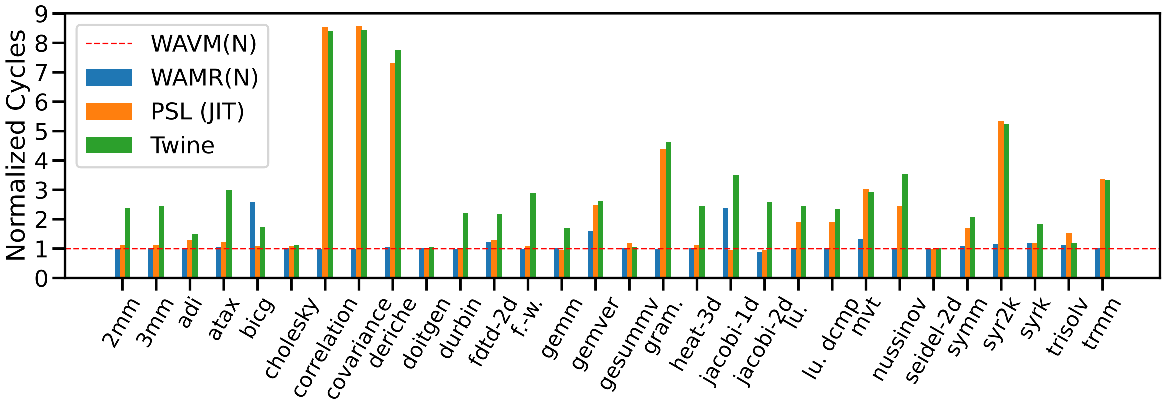


Figure 5. Execution Cycle Comparison on PolyBench between Twine[30] and PSL. All cycles are normalized by running PolyBench on WAVM (Native) without SGX. WAMR (the runtime of Twine) and WAVM (the runtime of PSL) demonstrate similar Native performance. However, PSL-JIT demonstrates up to 3.7 times latency improvement compared to Twine.

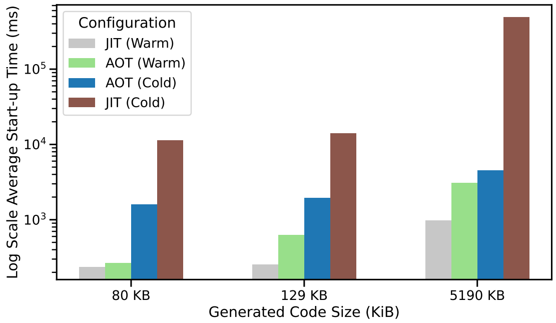


Figure 6. Startup Time Comparison of PSL Under different WASM Compilation Options and Cache Status The startup time includes the time of fetching the code and input from storage server and loading the code to enclave.

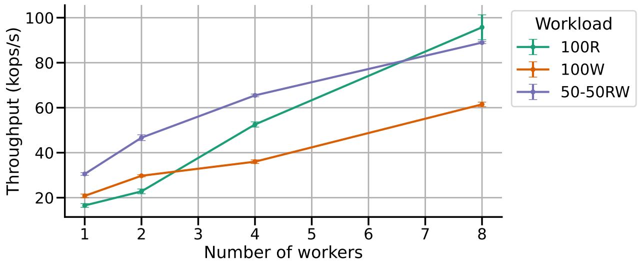


Figure 7. Throughput Scalability With The Number of Workers on Different Read-Write Ratios Each worker runs 6 application threads and signs for every 200^{th} block.

Release-consistent locking. Release-consistent locking mechanism gives a considerable overhead. The throughput of a single-threaded 1 worker system with no locking is around 17k ops/s. However, that with locking drops to 8k ops/s. This throughput remains almost fixed as we scale to 2, 4, and 8 workers as mutual exclusion only allows one worker to progress at a time. We therefore conclude that release-consistent locking should be used very rarely. We emulate single-threaded Paxos in our implementation and the throughput is 9.2k ops/s which is close to our throughput with release-consistent locking.

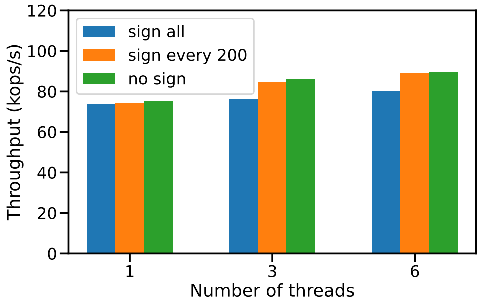


Figure 8. Overhead of Signatures vs Number of application threads in an 8 worker setup. "sign all" signs every block; "sign every 200" defers signature for every 200 blocks; "no sign" turns off both signature and verification, and only encrypts the blocks. The throughput overhead from signing all blocks to not signing is ~10%.

Cache performance. To understand the performance of the system with bigger application sizes, we increase the number of keys in the YCSB trace such the total YCSB application size becomes 256MiB. For this application we run with Memtable sizes of 64, 128 and 256MiB. We get throughput of 53.6k, 54k, and 63.2k ops/s respectively for a 50% write workload. Since SGX comes with 128MiB EPC size, using caches higher than that size does not scale well due to expensive paging operations.

Behavior under failure. To test the failure resilience of our system, we experimented with a faulty storage server that drops 2/3rds of all blocks it receives. This simulates a storage server with a lossy link. The time-series for the

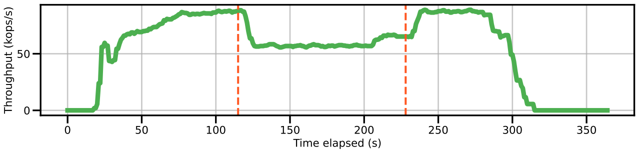


Figure 9. Throughput Timeline under transient storage server failures. The red lines denote the exact time one healthy storage server crashes and comes back up respectively. During the downtime the commits happen through 1 healthy and 1 lossy server, prompting multiple retries.

experiment is shown in Figure 9. In the beginning, we see that the throughput of the system faces no drops due to the lossy storage server, since the other two servers send the acknowledgments immediately. Then we kill one healthy storage server number and bring it back up after some time. During this phase, the throughput drops to almost half. But it smoothly recovers when the healthy storage server comes back up.

7.5 Case Study: Distributed and Privacy Preserving Deep Neural Network Training

We demonstrate the applicability and scalability by training a Deep Neural Network with multiple workers. Workers are connected by PSL with a shared memory paradigm. and how scaling multiple numbers of workers can finish computing long, intensive tasks, we see how PSL can scale to do secure deep neural network training.

We train a language model on Eurlex [9], that has more than 1 million parameters with ADAM optimizer [49]. We port the Sub-linear Deep Learning Engine (SLIDE)[11], a CPU-based deep learning algorithm that utilizes multi-threading to reduce training time. The original SLIDE reports better performance than training with GPUs. We implement the data loader that fetches the training set and the SLIDE code from the storage server. The parameters of the deep learning model, such as biases and weights, are exchanged through a shared memory array abstraction. The per-epoch latency is collected for 10 epochs. We also compare these benchmarks to compiling SLIDE with -O3 and running natively.

Figure 10 shows that PSL achieves almost linear scaling as we scale from 1, 2, 4, and 8 worker nodes on distributed training latency. Furthermore, PSL only introduces 2x overhead compared against running with an unsecured and unsandboxed native worker, despite (1) the overhead of publishing the training data and weights to the storage server, (2) the overhead of SGX in data confidentiality, integrity and isolation, and (3) the overhead of WASM sandbox, which provides dynamic safety checks[27].

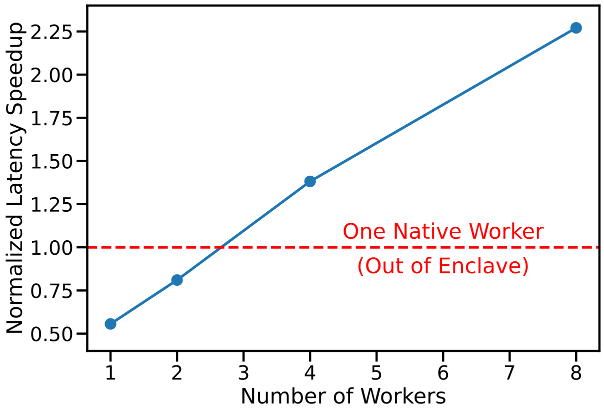


Figure 10. Normalized Latency Speedup of Distributed Deep Learning Training with increasing number of workers. PSL is only two times slower compared to a native worker with no enclave protection and WASM sandboxing. The speedup is linear with increasing number of workers in this domain.

8 Related Work

Current Frameworks for FaaS. There are existing stateless cloud-based FaaS implementations that support different forms of isolation. There is container based isolation like OpenWhisk [19] and OpenFaaS [34] that are stateless. There is also more lightweight forms of isolation like AWS Lambda [7] which is based on the Firecracker MicroVM [3]. Faasm [40] is the most similar to us as they use a WASM runtime engine, but do not run inside secure hardware. An earlier implementation, Secure Concurrency Layer [13], that exchanges states on a peer-to-peer network [12], only works for Intel SGX with limited programming language support.

Secure Execution with TEE. There is existing FaaS based secure hardware that focus on optimizing specific metrics like cold start latencies by reusing enclave [50] or enable scalable memory sharing to securely share FaaS runtimes [20, 29, 47]. In contrast, PSL attempts to build an entire FaaS framework with stateful computation. S-FaaS [5] and Occulum [38] proposes a FaaS framework on top of secure hardware. S-FaaS introduces transitive attestation similar to PSL and contributes a secure resource measurement to bill clients accurately using Intel TSX extensions. Occulum uses SFI to create light-weight enclaves, but only support stateless computation and also rely on specific Intel hardware extensions.

Consistency protocols on TEE. Consistency protocols on TEEs generally fall into two categories: (1) running the whole protocol inside enclave, e.g., CCF[24], (2) using enclaves as trusted endorsers, e.g., Nimble[6]. Although projects in the first category have larger TCB sizes, the confidentiality guarantees are also higher. Hence, PSL uses a similar approach.

A similar line of work tries to adapt known consensus protocols into TEEs. [44, 45]. TEEs provide the non-equivocation guarantee which helps CFT protocols give BFT guarantees. However, naively using this leads to easy responsiveness attacks. FlexiTrust protocols [22] solve this issue by using more nodes but provide better performance. However, all these works guarantee total order at the cost of parallelism. To the best of our knowledge, PSL is the first to formally consider performant eventual consistency in a TEE-based system with asynchronous network.

9 Conclusion and Future Work

In this work, we present PSL, a lightweight, secure and stateful FaaS framework in TEE that supports WASM with JIT compilation which scales to multiple workers a provide 95k ops/s write throughput. It ensures secure state persistence and scalable and efficient eventual state consistency. We show a case study of distributed training on confidential data with PSL to show its adaptivity and scalability. We leave it as future work to verify JIT transformations with a formal security guarantee. We leave addressing rollback attacks, provenance tracking, and long-term key storage as future works.

We leave it as future work to reduce cold start latency of JIT by WASM runtime swapping: upon a cold start, the code starts an interpreted version of WAMR, while the JIT engine compiles code in the background. In addition, we assume JIT engine is trusted in transforming WASM to machine code. To fully trust the transformations, one can create a JIT engine, such as [17] on creating verified binary lifters, that only transforms verified transformations in the code. In future work, we would also like to compare our performance against state-of-the-art TEE-based consensus protocols, such as MinBFT[44], FlexiBFT[22].

References

[1] [n. d.]. https://en.bitcoin.it/wiki/BIP_0032

[2] [n. d.]. Emscripten Compiler Frontend. https://emscripten.org/docs/tools_reference/emcc.html. Accessed: 2024-04-19.

[3] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 419–434. https://www.usenix.org/conference/nsdi20/presentation/agache

[4] Fritz Alder, N Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. 2019. S-faas: Trustworthy and accountable function-as-a-service using intel sgx. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*. 185–199.

[5] Fritz Alder, N Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. 2019. S-faas: Trustworthy and accountable function-as-a-service using intel SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*. 185–199.

[6] Sebastian Angel, Aditya Basu, Weidong Cui, Trent Jaeger, Stella Lau, Srinath Setty, and Sudheesh Singanamalla. 2023. Nimble: Rollback Protection for Confidential Cloud Services. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 193–208. https://www.usenix.org/conference/osdi23/presentation/angel

[7] AWS. [n. d.]. AWS Lambda. https://aws.amazon.com/lambda/, note = Accessed: 2021-05-1.

[8] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2018. A case for serverless machine learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS*, Vol. 2018. 2–8.

[9] Ilias Chalkidis, Manos Fergadiotis, Prodromos Malakasiotis, and Ion Androutsopoulos. 2019. Large-Scale Multi-Label Text Classification on EU Legislation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Florence, Italy, 6314–6322. https://doi.org/10.18653/v1/P19-1636

[10] K. Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.* 3, 1 (feb 1985), 63–75. https://doi.org/10.1145/214451.214456

[11] Beidi Chen, Tharun Medini, and Anshumali Shrivastava. 2019. SLIDE : In Defense of Smart Algorithms over Hardware Acceleration for Large-Scale Deep Learning Systems. *CoRR* abs/1903.03129 (2019). arXiv:1903.03129 http://arxiv.org/abs/1903.03129

[12] Kaiyuan Chen, Ryan Hoque, Karthik Dharmarajan, Edith LLontopl, Simeon Adebola, Jeffrey Ichnowski, John Kubiatowicz, and Ken Goldberg. 2023. FogROS2-SGC: A ROS2 Cloud Robotics Platform for Secure Global Connectivity. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 1–8.

[13] Kaiyuan Chen, Alexander Thomas, Hanming Lu, William Mullen, Jeffery Ichnowski, Rahul Arya, Nivedha Krishnakumar, Ryan Teoh, Willis Wang, Anthony Joseph, et al. 2022. SCL: A Secure Concurrency Layer For Paranoid Stateful Lambdas. *arXiv preprint arXiv:2210.11703* (2022).

[14] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (Abu Dhabi, United Arab Emirates) *(ASIA CCS '17)*. Association for Computing Machinery, New York, NY, USA, 7–18. https://doi.org/10.1145/3052973.3053007

[15] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[16] Intel Corporation. 2017. Intel(R) Software Guard Extensions SDK for Linux* OS. https://download.01.org/intel-sgx/linux-1.8/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.8_Open_Source.pdf.

[17] Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S. Adve, and Christopher W. Fletcher. 2020. Scalable validation of binary lifters. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 655–671. https://doi.org/10.1145/3385412.3385964

[18] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 205–220. https://doi.org/10.1145/1323293.1294281

[19] Karim Djemame, Matthew Parker, and Daniel Datsev. 2020. Open-source serverless architectures: an evaluation of apache openwhisk. In *2020 ieee/acm 13th international conference on utility and cloud computing (ucc)*. IEEE, 329–335.

[20] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2021. Scalable Memory Protection in the PENGLAI Enclave. (2021).

[21] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. 1990. Memory Consistency and

Event Ordering in Scalable Shared-Memory Multiprocessors. In *ISCA*. ACM.

[22] Suyash Gupta, Sajjad Rahnama, Shubham Pandey, Natacha Crooks, and Mohammad Sadoghi. 2023. Dissecting BFT Consensus: In Trusted Components we Trust!. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 521–539. https://doi.org/10.1145/3552326.3587455

[23] Gus Gutoski and Douglas Stebila. 2015. Hierarchical deterministic bitcoin wallets that tolerate key leakage. In *International Conference on Financial Cryptography and Data Security*. Springer, 497–504.

[24] Heidi Howard, Fritz Alder, Edward Ashton, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Antoine Delignat-Lavaud, Cédric Fournet, Andrew Jeffery, Matthew Kerner, Fotios Kounelis, Markus A. Kuppe, Julien Maffre, Mark Russinovich, and Christoph M. Wintersteiger. 2023. Confidential Consortium Framework: Secure Multiparty Applications with Confidentiality, Integrity, and High Availability. *Proc. VLDB Endow.* 17, 2 (oct 2023), 225–240. https://doi.org/10.14778/3626292.3626304

[25] Jeffrey Ichnowski and Ron Alterovitz. 2019. Motion Planning Templates: A Motion Planning Framework for Robots with Low-power CPUs. In *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*.

[26] Jeffrey Ichnowski, Kaiyuan Chen, Karthik Dharmarajan, Simeon Adebola, Michael Danielczuk, Vıctor Mayoral-Vilches, Hugo Zhan, Derek Xu, Ramtin Ghassemi, John Kubiatowicz, et al. 2022. FogROS2: An Adaptive and Extensible Platform for Cloud and Fog Robotics Using ROS 2. *arXiv preprint arXiv:2205.09778* (2022).

[27] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 107–120. https://www.usenix.org/conference/atc19/presentation/jangda

[28] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. 2021. SFI safety for native-compiled Wasm. In *Network and Distributed Systems Security (NDSS) Symposium*.

[29] Dayeol Lee, Kevin Cheang, Alexander Thomas, Catherine Lu, Pranav Gaddamadugu, Anjo Vahldiek-Oberwagner, Mona Vij, Dawn Song, Sanjit A. Seshia, and Krste Asanovic. 2022. Cerberus: A Formal Approach to Secure and Efficient Enclave Memory Sharing. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1871–1885. https://doi.org/10.1145/3548606.3560595

[30] Jämes Ménétrey, Marcelo Pasin, Pascal Felber, and Valerio Schiavoni. 2021. Twine: An embedded trusted runtime for webassembly. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 205–216.

[31] Microsoft. [n. d.]. OpenEnclave Switchless. https://github.com/openenclave/openenclave/tree/master/samples/switchless. Accessed: 2021-05-1.

[32] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 227–240. https://www.usenix.org/conference/atc18/presentation/oleksenko

[33] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*. 305–319.

[34] OpenFaaS. [n. d.]. OpenFaaS. https://www.openfaas.com/.

[35] Matthias J. Reisinger. 2023. PolyBenchC-4.2.1. https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1.

[36] Mark Russinovich, Edward Ashton, Christine Avanessians, Miguel Castro, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Cédric Fournet, Matthew Kerner, Sid Krishna, et al. 2019. CCF: A framework for building confidential verifiable replicated services. *Microsoft, Redmond, WA, USA, Tech. Rep. MSR-TR-2019-16* (2019).

[37] Andrew Scheidecker and other contributors. 2024. WAVM: WebAssembly Virtual Machine. https://github.com/WAVM/WAVM Accessed: 2024-04-19.

[38] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and efficient multitasking inside a single enclave of intel sgx. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 955–970.

[39] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. https://doi.org/10.14722/ndss.2017.23193

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

[41] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing Page Faults from Telling Your Secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security* (Xi'an, China) *(ASIA CCS '16)*. Association for Computing Machinery, New York, NY, USA, 317–328. https://doi.org/10.1145/2897845.2897885

[42] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592* (2020).

[43] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 645–658.

[44] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. 2013. Efficient Byzantine Fault-Tolerance. *IEEE Trans. Comput.* 62, 1 (2013), 16–30. https://doi.org/10.1109/TC.2011.221

[45] Weili Wang, Sen Deng, Jianyu Niu, Michael K. Reiter, and Yinqian Zhang. 2022. ENGRAFT: Enclave-guarded Raft on Byzantine Faulty Nodes. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 2841–2855. https://doi.org/10.1145/3548606.3560639

[46] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 81–93.

[47] Zhijingcheng Yu, Shweta Shinde, Trevor E Carlson, and Prateek Saxena. 2020. Elasticlave: An Efficient Memory Model for Enclaves. *arXiv preprint arXiv:2010.08440* (2020).

[48] Miao Zhang, Yifei Zhu, Cong Zhang, and Jiangchuan Liu. 2019. Video processing with serverless computing: A measurement study. In *Proceedings of the 29th ACM workshop on network and operating systems support for digital audio and video*. 61–66.

[49] Zijun Zhang. 2018. Improved adam optimizer for deep neural networks. In *2018 IEEE/ACM 26th international symposium on quality of service (IWQoS)*. Ieee, 1–2.

[50] Shixuan Zhao, Pinshen Xu, Guoxing Chen, Mengya Zhang, Yinqian Zhang, and Zhiqiang Lin. 2023. Reusable enclaves for confidential serverless computing. In *Proceedings of the 32nd USENIX Conference on Security Symposium* (Anaheim, CA, USA) *(SEC '23)*. USENIX Association, USA, Article 225, 18 pages.