Confidential Serverless Made Efficient with Plug-In Enclaves

Mingyu Li, Yubin Xia, Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai AI Laboratory, Shanghai Jiao Tong University
{maxullee,xiayubin,haibochen}@sjtu.edu.cn

***Abstract*—Serverless computing has become a fact of life on modern clouds. A serverless function may process sensitive data from clients. Protecting such a function against untrusted clouds using hardware enclave is attractive for user privacy. In this work, we run existing serverless applications in SGX enclave, and observe that the performance degradation can be as high as 5.6× to even 422.6×. Our investigation identifies these slowdowns are related to architectural features, mainly from page-wise enclave initialization. Leveraging insights gained from our overhead analysis, we revisit SGX hardware design and make minimal modification to its memory model. We extend SGX with one new primitive—region-wise *plugin enclaves* that can be mapped into existing enclaves to reuse attested common states amongst functions. By remapping plugin enclaves, an enclave allows in-situ processing to avoid expensive data migration in function chains. Experiments show that our design effectively reduces enclave serverless latency by 94.74-99.57%, and boosts the autoscaling throughputs by 19-179×.**

***Index Terms*—Intel SGX, Serverless, Confidential Computing**

I. Introduction

Serverless computing is becoming the next generation of cloud computing, including Amazon AWS Lambda [1], Microsoft Azure Functions [2], Google Cloud Functions [3], etc. Serverless computing enables developers to concentrate on the business logic by writing fine-grained, simple and standalone functions, with minimal concerns on the deployment, management, scalability issues, etc. That is why serverless gains its another popular name——Function-as-a-Service (FaaS). Typically, a serverless function is event-driven, either via a user HTTP/HTTPS request, or another function's invocation—in a chained way. Real-world serverless characterization [4] reported that 54% of the serverless applications contain only one function, and 50% of the functions execute less than 1 second; hence serverless applications are extremely sensitive to service latency. Both industry and academia have been improving the performance of serverless computing [5]–[10].

Serverless applications might process privacy-sensitive workloads. According to use cases from Amazon AWS [11] and Google Functions [12], serverless computing can be used for security-critical or privacy-sensitive applications such as Auth0 (authentication) [13], Alexa chatbot (users intentions) [14], face recognition (bioinformation) [15], etc. There is a demanding need to protect user privacy on such a complex cloud environment, from vulnerable cloud software, malicious co-located tenants or even possibly a suspicious cloud insider. Architectural support for trusted execution environment (TEE), e.g., Intel SGX [16], [17], can provide secure *enclaves* which are fully isolated from the rest of the system, while allowing a remote user to attest the enclave's identity. Thus, TEE is considered as a promising technology to realize practical privacy-preserving serverless applications [10], [18]–[20].

However, existing TEE designs cannot well fit the serverless workloads. We first port five real-world privacy-critical serverless workloads (Table I) into a state-of-the-art enclave framework—Graphene-SGX [21], and observe a significant performance degradation from 5.6× to 422.6×. Using detailed performance profiling, our investigation shows that the majority of the overhead stems from the enclave initialization: both hardware enclave creation and attestation measurement generation dominate an enclave function's startup, ranging from 92.3% to 99.6%. The results also indicate another performance degradation factor, which arises from the secret data transfer between functions, occupying 15.0% to 34.7% of the execution time; it becomes even worse for long-chain function invocations. Although we apply software-based optimization proposed by previous studies [10], [22]–[24], the end-to-end latency of invoking an enclave function is still far from satisfactory. We further revisit Intel SGX hardware design, and conclude that the root cause for the inefficient enclave-based serverless is that current SGX design (both SGX1 [16] and SGX2 [17]) disables memory sharing between enclave instances. This share-nothing design offers strong security guarantee, but incurs significant startup latency which is unsuitable for today's serverless computing.

This paper presents a novel and flexible enclave abstraction, called PIE, to make confidential serverless computing efficient and practical. PIE proposes a new hardware memory primitive: *shared enclave region*, which can be immutably mapped into different isolated enclaves to achieve secure sharing. Taking advantage of this hardware primitive, the enclave-based serverless platform can create two types of logical enclaves: *plugin enclaves* and *host enclaves*. A plugin enclave fully consists of shared enclave region(s), and can contain language runtimes (e.g., Python), frameworks (e.g., Tensorflow), third-party libraries (e.g., OpenSSL library) or common states (e.g., machine learning models). Host enclaves, as with the current SGX design, are strictly isolated from each other, but can map plugin enclaves into their own enclave address space, in order to reuse plugin enclaves' loaded contents as well as their readily generated measurements. A host enclave can further remap plugin enclaves to adapt to different application logics without migrating its secret data. PIE's key improvement to efficiency is that PIE-based mapping is region-wise instead of page-wise, and host enclaves can invoke plugin enclave' procedures via lightweight function calls (∼60 cycles).

PIE design is fully compatible with SGX1 and SGX2 semantics, and reuses most of the existing hardware design to minimize its implementation complexity. PIE's architectural extension to SGX includes a new page type to indicate shared enclave memory, and two new instructions EMAP and EUNMAP to map/unmap a plugin enclave to/from a host enclave. To ensure the consistency between the content and measurement of a plugin enclave, PIE reuses SGX2 dynamic resizing to implement a hardware-enforced copy-on-write mechanism.

To show how PIE improves serverless workloads, we use a real SGX-enabled cloud machine, and emulate PIE instructions by adding cycle-accurate latency to EMAP/EUNMAP operations. We partition the common serverless infrastructure such as language runtimes, third-party libraries, and user functions in plugin enclaves, and secret data in host enclaves. For input/output dataflow amongst serverless functions, we remap plugin enclaves to avoid secret data migration. Evaluation results show that PIE can reduce 94.74-99.57% function startup latency, achieve 19-179× throughput boost in function autoscaling, and 16.6-20.7× speedup in data transfer of function chaining. Moreover, PIE-based serverless can scale up to 10× enclave instances density than current SGX hardware.

The contributions of this paper are summarized as follows:

- We present the first quantitative study on the performance of real-world serverless applications protected in SGX enclaves, and identify the root cause of performance degradation mainly lies in the current SGX design.
- We describe PIE design and its minimal extension to SGX, which is feasible for real hardware implementation.
- We show that enclave serverless workloads can benefit from PIE which reduces 94.74-99.57% startup latency and improves throughputs by a factor of 19×-179×.

II. Background

A. Intel Software Guard Extension (SGX)

Since the 6th generation of Intel processors, Intel introduces a novel security extension, named Software Guard Extension (SGX), which allows to create a user-level *enclave* embedded in a process. SGX CPU isolates an enclave from a wide range of threats, including (1) direct memory access (DMA) from peripheral devices, (2) privileged system software like firmware, operating system and hypervisor, (3) the application that co-locates with the enclave in the same address space, (4) enclaves that share the same hosted application where one's faults or bugs cannot compromise the other.

Enclave Access Control Model. Enclave memory is named Enclave Page Cache (EPC). SGX has a strict access control model on EPC: *an EPC only belongs to one enclave instance*. Each enclave instance has a unique Enclave Identifier (EID) stored in its SGX Enclave Control Structure (SECS). When an EPC page is added to an enclave, SGX associates this EPC page with a metadata named EPC Map (EPCM). An EPCM entry indicates the page type, permission, virtual address (VA) as well as the owner EID for an EPC page, as shown in Figure 1. When an executing enclave tries to access a particular EPC page, CPU will check whether its SECS.EID matches the corresponding EPCM.EID to this EPC page. Enclave metadatas such as SECS and EPCM are inaccessible to any software.

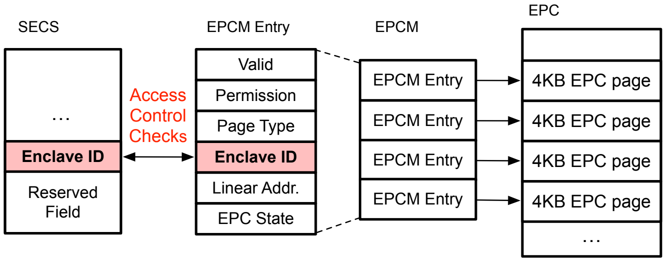


Fig. 1: **SGX memory access control model.** An enclave instance can only access an EPC page whose EPCM.EID equals to its own SECS.EID.

Remote and Local Attestation. To prove that an enclave is correctly established, SGX provides a hardware mechanism for a remote enclave user to attest its identity. During the launch process, CPU computes a SHA-256 hash by measuring each EPC page, and finalizes this measurement value in a hardware-protected register. Any tampering with this process will result in a different measurement. SGX provides another efficient local attestation for enclaves on the same CPU to identify each other, thus establishing mutual trust.

B. Serverless Computing

Serverless computing is a rapidly growing cloud application model [1]–[3], [25]. Serverless computing offers developers an advantage of writing fine-grained, simple and standalone functions. Functions can be organized into a chain for processing composition. A serverless application is an event-driven, request-oriented interactive service, and desires low latency and high throughput. To match the invocation rate, serverless platforms automatically scales the function instances to be executed in parallel on available resources.

Serverless Latency Optimization. Since a function only handles a specific piece of logic, users expect low latency for these services. According to real-world serverless characterization on Azure Functions [4], 54% of the serverless applications only contain one function, and 50% of the functions take less than 1 second execution time on average. Many works have proposed optimizing the latency of the serverless startup [5]–[7], [9].

III. Motivation

Traditional serverless platforms leverage containers [26], [27] or virtual machines [7], [28] to confine a function into a sandbox. Their threat model is to protect the cloud from untrusted function executions and protect a function from co-locating tenants. Quite the opposite, SGX enclave adopts a reverse sandbox, which uses hardware to prevent the cloud from inspecting or interfering the sensitive computation. As Intel SGX can protect a user-level workload from an untrusted environment, it is a good fit to protect cloud sensitive processing, especially confidential serverless computation workloads.

TABLE I: The list of privacy-critical serverless applications we used or repurposed as benchmarks.

Application	Description	Language Runtime	Major Libraries Used	Total Libs.	Code + Read-Only Data Size	Data Size	Heap Size
auth	login authentication	NodeJS14.15	basic-auth, tsscmp, passport	6	67.72MB	0.23MB	1.85MB
enc-file	cloud storage encryption	NodeJS14.15	libicuata, libicui18n, crypto	13	68.62MB	0.23MB	1.90MB
face-detector	facial image recognition	Python3.5	Tensorflow, Numpy, OpenCV	53	66.96MB	2.38MB	122.21MB
sentiment	textual sentiment analysis	Python3.5	Numpy, Scipy, NLTK, Textblob	152	113.89MB	5.61MB	19.34MB
chatbot	personal voice assistant	Python3.5	Tensorflow, Pandas, llvmlite, sklearn	204	247.08MB	9.53MB	55.90MB

In our test, we modify the entry point of a function to invoke a function image protected within an enclave. The workflow of an enclave serverless instance is depicted in Figure 2.

Protecting serverless functions with SGX enclaves raises several concerns with respect to performance:

- **What is the overhead introduced to a serverless instance startup?** As the creation time of an enclave is now accounted in a function startup latency, it is important to understand its impact since serverless workloads typically requires low latency. Figure 3b describes the startup breakdown of real-world serverless workloads we evaluated. Particularly, Table II measures the SGX instruction latencies of our microbenchmarks.
- **How does serverless autoscaling perform when being secured in enclaves?** A promising feature of serverless computing is that the cloud vendor offers instance autoscaling which is corresponding to the invocation rate. For the purpose of high utilization, cloud vendors desire high throughputs. Figure 4 measures the latency distribution of enclaves functions when serving 100 concurrent requests.
- **What is the overhead of an enclave-protected function chain?** In the function chain mode, data must be transferred between each function. Figure 3c shows the cost of transferring secret data between enclave instances.

A. Quantitative Evaluation

Experimental Setup. We run all experiments on an Intel NUC7PJYH PC [1], with Pentium Silver J5005 at 1.50GHz with 2 hyper-threaded cores (totally 4 logical cores), 16GB DDR4, and 128MB processor reserved memory (≈94MB EPC) on Ubuntu Server 18.04 LTS, Linux kernel 5.4.0, Intel SGX SDK 2.12 and microcode version 0xd6. CPU frequency scaling governor was set to the "performance" mode , and dynamic frequency and voltage scaling were disabled during experiments.

Measuring Methodology. We access the CPU timestamp counter via RDTSCP to estimate the execution time in CPU clock cycles. On SGX2, RDTSCP can be executed within the enclave for accurate measurement. To best eliminate measurement errors, we run each group of test for 1,000 runs. To minimize the context switches overhead such as Asynchronous Exit (AEX), we set CPU affinity to SGX threads and route I/O interrupts to non-SGX threads.

TABLE II: SGX instructions latency (in cycles) on our testbed.

SGX1 Creation Instruction	Median Latency	SGX2 Creation Instruction	Median Latency	Other Instruction	Median Latency
ECREATE	28.5K	EAUG	12K	EREMOVE	4.5K
EADD	13K	EMODT	6K	EGETKET	40K
EEXTEND	5.5K	EMODPR	8K	EREPORT	34K
EINIT	88K	EMODPE	9K	EENTER	14K
		EACCEPT	5.5K	EEXIT	6K

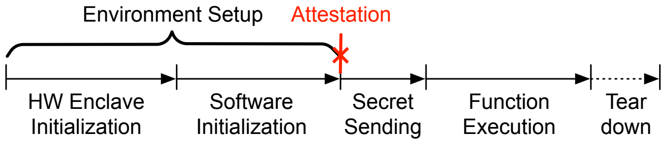
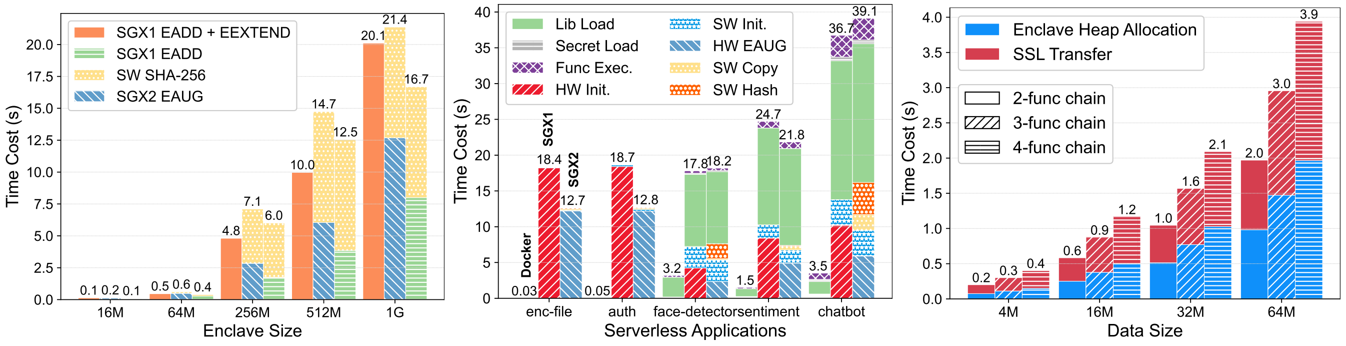


Fig. 2: **The workflow of an SGX enclave serverless instance.** A user must attest the enclave's environment before sending the secret data to the enclave for secure processing. Software initialization includes the loading time of language runtime, framework, and third-party libraries. The function execution may involve dynamic module/package loading. The end-to-end latency only comprises solid arrows.

Measuring SGX Instructions and Enclave Startup Latency. SGX instructions cannot be measured using a loop, it must be executed by following a legitimate order and the specification. We record the duration of each instruction in a memory buffer, and calculate their median cycles as shown in Table II.

In SGX1, we observe two main performance factors during enclave creation: (1) hardware enclave creation and (2) enclave measurement generation. For (1), CPU must use EADD to add every page with specified permission and virtual address (VA). This not only allocates an EPC page for the target enclave, but also requires updating the corresponding EPCM entry to track this EPC. From Intel Software Developer Manual (SDM) [29], EADD disallows concurrent addition to the same enclave instance, since a concurrent model increases the hardware formal verification complexity [17]. For (2), CPU needs to generate a SHA-256 based measurement, initialized from ECRATE, to EADD and EEXTEND on each EPC page, and finalized by EINIT. During this process, the most expensive one is EEXTEND, which takes 5.5K cycles to measure a 256-byte chunk of EPC data at one time. To measure a whole EPC page, it takes around 88K cycles in total. Our experiments shows that software-based SHA-256 measurement from OpenSSL is much more efficient, only 9K cycles for an EPC. Due to

[1] Until Nov. 2020, NUC7PJYH and NUC7CJYH are the only two commercially available machines we can find that support SGX2 instructions.



(a) Enclave instance startup time breakdown. (b) Startup breakdown of enclave functions. (c) Data transfer cost between enclaves.

Fig. 3: **Measuring serverless functions in SGX enclaves.** In (a), the first two columns shows the latency breakdown of pure SGX1 EADD and pure SGX2 EAUG enclave creation, respectively. The third column leverages optimization of combining SGX1 EADD and software-based SHA-256. In (c), the data trasfer cost becomes dominant when the function length increases.

the lack of implementation detail, a possible reason is that hardware EEXTEND performs intensive validation on each update. Advanced vector extensions (AVX) or streaming SIMD extensions (SSE) may be beneficial to accelerate EEXTEND.

In SGX2, an enclave allows to dynamically grow or shrink its size using EAUG -> EACCEPT flow. This flow is particularly efficient for heap-intensive workloads, as demonstrated in [10]. For code-intensive workloads, it requires further software-based measurement and update the EPC permission from "rw-" to "r-x" via EMODPR. EMODPR requires one more EACCEPT to verify the updated page permissions.

The startup latency of SGX1 and SGX2 enclaves is depicted in Figure 3a. Although SDM [29] does not explicitly state the concurrent restrictions on EAUG, our test shows that it is also impossible to expand an enclave using concurrent EAUG, because EAUG needs to update the same SECS page and therefore forbids concurrent modifications.

Measuring SGX Serverless Startup. We select five representative real-world serverless examples from Amazon AWS [11]. The first two examples consider login authentication and cloud encryption, which protects client's credentials and encryption keys. The rest consider machine learning inferences on either human text, image, or voice which can also be sensitive. They are typically user-facing, and thus requires low latency. We observe that today's serverless developers tend to use high level languages, and library OS is a good choice to ease the porting. Hence we use a state-of-the-art SGX library OS, Graphene-SGX v1.1 [21], and integrate it with an open-source serverless platform Fn [27] to run these serverless applications. In particular, we focus on two language runtimes, NodeJS and Python, because they are the two most popular languages used by commodity serverless platforms [30], [31].

Figure 3b shows the startup latency of a serverless function in (1) a native docker environment, (2) SGX1 enclave environment, and (3) SGX2 enclave environment. We observe a significant performance overhead introduced by enclave protection, varying from 5.6*times* to 422.6*times*. Both enc-file and auth applications are heap-intensive workloads, in which we found that SGX2 EAUG is more efficient than SGX1 EADD, and saves 31.9% startup cost. An advantage of EAUG is that EAUG automatically zeros out the EPC content for the enclave. For code-intensive workloads (sentiment and chatbot), we observed that SGX2 performs worse than SGX1 because SGX2 enclaves have to copy the runtime code and update the EPC page permission using EMODPR (8K cycles) and EACCEPT (5.5K cycles). Another dominant performance degradation factor is the process of third-party library loading, which is 5x to 13x slower than native docker environment, and can even occupy more than 55% startup time. The main reason is that loading tens to hundreds of libraries requires frequent context switches (namely ocalls), as studied in [22], [23].

Surprisingly, we found that 4 out of all 5 enclave functions execution time were still below 1s. The only exception is chatbot (3.02s), which incurs 19,431 ocalls to read the external files when generating the echo speech. After applying the fast interface optimization from HotCalls [23], the function execution time can be optimized to 0.24s.

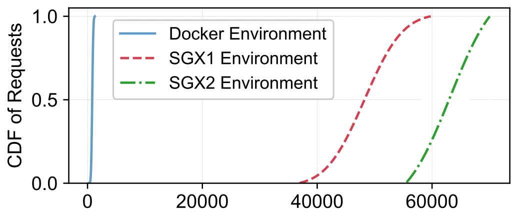


Fig. 4: **End-to-end latency (ms) of face-detector.** This test measures the latency distribution of 100 concurrent requests.

Measuring SGX Serverless Autoscaling. Serverless platforms offer function autoscaling when serving concurrent requests. For locality purposes, it is possible to scale up the same function instances on a single machine if resources permit. For all five functions, we increase the invocation rate per minute to test the autoscaling. Unfortunately, it is impossible to run more than 30 enclave functions on our testbed, which is limited to the capacity of physical memory (16GB), whereas

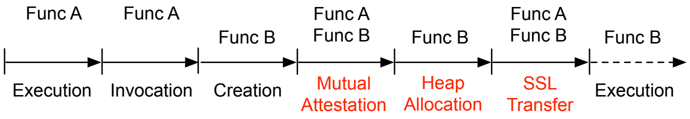


Fig. 5: **The steps of secret data transfer between two serverless enclave functions.** The SSL transfer includes data marshalling and unmarshalling, two data copies, in addition to AES-128-GCM encryption and decryption.

the native Fn platform [27] can serve more than 100 Docker-based requests concurrently.

When setting the hard limit of enclave instances to 30, we observe prolonged tail latencies of enclave functions, as shown in Figure 4 (due to space limit, we only show the face-detector application, the latency distributions of other serverless applications are alike). Since there are only 94MB physical EPCs available, concurrent enclave startups lead to extremely high EPC contentions, which even degrade the performance of other concurrently running enclaves. The performance penalty can even be 11× (from original 18.4s to 202.0s).

Measuring SGX Serverless Function Chain. In a function chain, secrets need to be transferred across enclave boundaries. We conduct a microbenchmark where we vary the data transfer size between functions. These procedures include (i) mutual attestation between function A and function B, (ii) SSL handshake between function A and function B, (iii) function B allocates a heap big enough to accommodate the secret data, (iv) transferring secret data between function A and function B (double copy and data en/decryption). We illustrate this procedure in Figure 5 (step (ii) omitted). We deem the sub-step (i) and (ii) to be constant-time (less than 25ms on our testbed), and show the cost of (iii) and (iv) in Figure 3c. The overhead of in-enclave heap allocation exceeds SSL transfer when the data size reaches 94MB because of the expensive EPC eviction overhead [22], [24], [32].

For small message passing between functions, the cost of secret transfer can be negligible (almost within 1s), compared with enclave initialization time (varying between 12s and 29s). For large message passing (>32MB) between enclaves, the data migration can be a bottleneck. Figure 3c shows that two massive data communication scenarios: (1) the secret data that requires transferring between functions is massive. (2) the secret data needs to travel across many serverless functions in a chained model. In both cases, moving secrets is quite expensive. Industrial statistics [4] reported that a real-world chain can be as long as 10 functions, which dramatically amplifies the impact of secret data transfer.

B. Software Optimization

To optimize these serverless applications, we have thought of the following solutions:

Template-based Start. Since the startup involves multiple rounds of loading third-party shared libraries, which inevitably incurs significant `EENTER`/`EEXIT` overhead, we can construct a template image containing all needed states, and set the entry point to the first line of user logic, as explored in [7], [8]. Using this technique, the enclave library loading time with 152 libraries (114MB in total) can be optimized from 13.53s to 1.99s (6.8x). However, the hardware enclave creation (between 4.2s and 18.2s) still dominates the startup latency.

Reuse-based Start. A conventional startup latency optimization is to reuse an existing instance instead of creating one from scratch. This is called warm-start in serverless [4]. In the context of enclave, an environment reset is a must in case of information leakage of the last function, or environment damage that compromises the next function, as discussed in [33]. Despite a feasible optimization, reuse-based start does not solve the latency issue of autoscaling when concurrent requests exceeds the pre-warmed instances, and inevitably occupies considerable resources.

Sharing-based Start. As discussed in [7], [9], sharing is key to efficient serverless computing. Unfortunately, this is impossible since SGX hardware design disables sharing EPC between enclaves. Another optimization idea is to run multiple functions sharing the same languages runtime in a single address space. Because different users are mutually distrusted, using the same address space requires in-enclave isolation. Prior studies [33], [34] achieved software-based in-enclave isolation relying on either compiler-based instrumentation or runtime-based confinement, introducing substantial amount of code into trusted computing base (TCB). Software-based isolation can be a suboptimal solution since an enclave user would prefer only trusting hardware for security guarantees.

C. Lessons Learned

Insight 1: Hardware `EADD` and software hashing achieves the fastest enclave startup. Our microbenchmark results shows that *SGX2 EAUG is no better than SGX1 EADD for code-intensive workloads*. To load a code page, SGX2 `EAUG` requires further `EMODPR` and `EACCEPT` procedures, introducing more 11.5K cycles. A more efficient way to initialize an enclave is to use SGX1 EADD and software-based SHA-256, as shown in the third column of Figure 3a, and use hardware/software co-design measurement. We also found that Intel SGX SDK uses expensive `EEXTEND` to measure initial heap pages allocated by `EADD`, which can be securely optimized via software zeroing before use (e.g., C library *calloc()*), which saves 78.8K cycles for an EPC page.

Insight 2: Common serverless states are non-sensitive and can be shared between functions. SGX assumes all EPC contents belonging an enclave are private. However, in the context of serverless computing, we found that the function environment, including the language runtimes, third-party libraries, and even the function itself are quite often open-source and contain no secrets. Typically, functions written in high-level languages such as Python, Go, Java, JavaScript, etc., bring a relatively heavyweight runtime, and sharing this runtime helps release a lot of memory pressure. Unfortunately, sharing EPC amongst enclave instances is not supported in current SGX hardware. An enclave function must always be initialized from scratch. With software-based optimization, enclave initialization and template-based library loading still incurs 12.25s latency for an 800MB enclave.

Insight 3: Transferring secret data between serverless functions is expensive. From our benchmarks, we found that serverless function chains involve abundant heap allocation and expensive data communication. Particularly, the data copying and SSL en/decryption dominate when the data size is smaller than physical EPC capacity (94MB on our testbed), whereas in-enclave heap allocation becomes influential for >94MB secret data because of additional EPC evictions. EPC evictions involve hardware re-encryption of paging-out contents and incur inter-processor interrupts (IPIs) for inter-thread synchronization. An ideal solution is to share the same secret data amongst different functions in order to remove data transfer bottlenecks, so-called *in situ* processing. This is also not possible in current SGX hardware.

Leveraging these insights, we conclude that enclave-protected serverless functions can benefit from secure sharing. Current SGX hardware does not support this. This motivates us to extend current SGX design to PIE for better efficiency.

IV. PIE DESIGN

A. Overview

PIE extends SGX design to support an efficient and flexible enclave memory model. Current SGX memory model provides private EPC which can only be accessed by one enclave instance. PIE introduces another hardware primitive: *shared EPC*. This primitive allows an enclave developer to build two kinds of logical enclaves: a *plugin enclave* made of shared EPCs that can run language runtimes, frameworks, shared libraries, or contain initial common states that can be reused by other enclaves, and a *host enclave* made of private EPCs runs a secure sandbox that processes the user secret (usually from a secure input channel), and carefully protects the processing and its final results. It is intuitive that many host enclaves can share common plugin enclaves for both spatial and temporal efficiency, by reusing the readily loaded contents and avoiding costly measurement generation. To this end, a PIE plugin enclave can be mapped into multiple host enclaves, as long as its measurement is verified by the recipient host enclave.

To support efficient mapping, PIE introduces a new instruction: `EMAP`. Unlike existing page-wise SGX instructions, which manipulate only one EPC page at a time, `EMAP` is a region-wise instruction that allows the recipient host enclave to access the whole virtual address space of the plugin enclave. To selectively unmap unnecessary libraries, PIE offers `EUNMAP` to reclaim the region of allocated virtual address space. This also offers an opportunity to eliminate expensive data transfer bottlenecks through remapping plugin enclaves of different application logics, while preserving the secret data to be processed in place. To retain security properties, PIE blocks write attempts to plugin enclaves, and uses a hardware-enforced copy-on-write mechanism to ensure the consistency between the measurement and contents of a plugin enclave. Copy-on-write is also desired when sharing initial states of plugin enclaves can help improve the performance. In essence, PIE enables enclave memory with normal DRAM operations (i.e., dynamic-mapping and copy-on-write). PIE proves that this extension is practical to real-world applications, and can benefit low latency of enclave applications without sacrificing desired security guarantees.

Particularly, in this paper, we apply PIE enclave model to confidential serverless computing. We discuss more optimization opportunities for other workloads in § VIII-B.

B. Threat Model

PIE follows the threat model of current SGX: all in-enclave code and data are trusted, including those in plugin enclaves. PIE's goal is not to reduce trusted computing base (TCB), but to improve performance. Indeed, we have considered another threat model that plugin enclaves are untrusted and able to isolate heavyweight runtimes and libraries, but this requires extensive software modification. Prior proposals of in-enclave isolation (e.g., [35]) are incompatible to interpreted languages (e.g., NodeJS and Python) that are widely used in serverless computing (see discussion in § VIII-A)). In contrast, PIE aims to achieve better compatibility and practicability.

PIE's root of trust is provided by hardware vendors using a verifiable measurement. Users must remotely attest the state (code and data included) of a host enclave before sending secret data. PIE leverages a *trust chain* model (see Figure 7), where the host enclave is responsible to locally attest all the used plugin enclaves to provide the whole-enclave security guarantee. Any incorrect interference with plugin enclave management by privilege software can be detected or aborted.

Architectural side-channels (e.g., Spectre [36], L1TF [37]) and CPU bugs (e.g., power-based fault injection [38]) are out of scope. They can be addressed by improving the processor's internal circuit design, or mitigated by updating corresponding microcode, which are orthogonal to this work. Denial-of-service is not considered because remote users can easily detect the unavailability of the cloud services.

C. New Metadata and New Instructions

As aforementioned (§ II), the SGX enclave access model is determined by SECS and EPCM: an enclave that holds an 8-byte enclave identifier (EID) in SECS can access an EPC page whose EPCM has the same EID. We hence extend the SECS of a host enclave to store the additional EIDs for plugin enclaves. To interact with the plugin enclaves, PIE introduces two new instructions: `EMAP` and `EUNMAP`. PIE's ISA extension is fully compatible with SGX1 and SGX2 semantics; an SGX1 or SGX2 enclave image can run on the PIE CPU without modification.

`EMAP`. `EMAP` adds an initialized plugin enclave EID to the SECS structure of the current host enclave. CPU checks if the intended virtual address range conflicts with the used address range. If virtual address range conflicts, `EMAP` will fail.

`EUNMAP`. As a reversed operation to `EMAP`, `EUNMAP` removes the specified plugin enclave EID from the SECS structure. After all intended `EUNMAP`s, the enclave software should invoke `EEXIT` to flush the stale TLB mapping.

Concurrency Restrictions. As with EADD and EAUG, PIE forbids concurrent execution of EMAP/EUNMAP in case of race condition on updating the same SECS data structure. PIE strictly follows the SGX linearizability model [39].

Instruction Privilege Considerations. Intel SGX instructions can be categorized into two groups: supervisor-mode (ENCLS) and user-mode (ENCLU). We choose EMAP/EUNMAP as user-mode for the following reasons: (1) Only the host enclave knows which plugin enclave should be mapped after local attestation. If we grant this right to the kernel, an untrusted kernel can map a malicious plugin enclave which may dump secret data to the unprotected memory. (2) To guard from kernel's unintended mapping if supervisor-mode EMAP/EUNMAP were allowed, the CPU should introduce two more instructions, e.g., EMAP_ACK/EUNMAP_ACK, to verify whether the kernel has performed the correct mapping as expected. More instructions as well as user/kernel interactions may add more complexity to both hardware and software implementations.

Hence, in PIE, EMAP/EUNMAP are designed to be user-mode instructions. An optimization opportunity is that a host enclave can batch all EMAP operations to include all wanted plugin enclaves without exiting the enclave mode, and switches to OS once, and then the OS updates all required page table entries (PTEs) also in a batch to improve efficiency.

D. New EPC Page Types

Shared EPC and Plugin Enclave. Each EPCM entry has an 8-bit PAGE_TYPE field, so we add a PT_SREG page type to indicate an shared EPC. CPU automatically masks the write permission bit for shared EPC pages. A shared EPC can be added to form a plugin enclave via EADD. EINIT must be used to complete the creation process because it finalizes the measurement of the plugin enclave. Once being EINIT'ed, a plugin enclave can be mapped to a host enclave via EMAP. SGX2 instructions are prevented from being applied to a plugin enclave, because these instructions will change its contents (EAUG), permissions (EMODPR) and page type (EMODT) after initialization. In a sense, a plugin enclave can be considered as an immutable enclave region. PIE considers this design choice for two security reasons: (1) Both SGX1 and SGX2 does not update the measurement once the enclave is initialized. PIE aims to remain backwards compatible with this semantic. It is insecure for a user to attest an enclave using a stale measurement. (2) Since a plugin enclave might have been mapped to multiple host enclaves, updating its contents may compromise the security guarantees of host enclaves.

Private EPC and Host Enclave. The property of a mutable EPC is the same as that of current SGX design. A private EPC only belongs to an enclave instance at any time. A host enclave can be composed of both private EPCs and shared EPCs, the latter of which are from mapped-in plugin enclaves. Any enclaves that contain a private EPC is deemed as a host enclave, and cannot be mapped to other enclaves.

When a host enclave attempts to write the contents of an shared EPC from its plugin enclave, CPU will trigger a page fault, and require OS to insert a private EPC at the faulting address via SGX2 EAUG. The host enclave then issues SGX2 EACCEPTCOPY to atomically copy the contents and permissions from the shared EPC to the newly EAUG'ed private EPC. Such a hardware-enforced copy-on-write mechanism protects the integrity of shared plugin enclaves.

TABLE III: **PIE's EPC page types.** PT_SREG is added to support shared EPCs and compose plugin enclaves.

Page Type	Allocated By	Contents
PT_SECS	ECREATE	Enclave Control Structure
PT_VA	EPA	Version Array
PT_TRIM	EREMOVE	Trimmed State
PT_TCS	EADD/EAUG	Thread Local Storage
PT_REG	EADD/EAUG	Private Regular Page
PT_SREG	EADD	Shared Immutable Page

E. Lifecycle

Plugin Enclave. The creation of a plugin enclave is identical to current enclave creation procedure: the enclave control structure (SECS) is created via ECREATE, and its memory contents are loaded via EADD. Both EADD and EEXTEND measures each shared EPC page, and EINIT finalizes the measurement generation as well as the plugin enclave creation. After performing EINIT, a plugin enclave is ready to be mapped to other host enclaves via EMAP.

A plugin enclave rejects further EAUG operations since this will result in an inconsistency between its contents and its measurement. Likewise, EREMOVE to a plugin enclave is only allowed when no host enclaves are using it (an EUNMAP from host enclaves must be executed). If an EREMOVE is successfully executed on the plugin enclave, CPU then disallows any EMAP to this plugin enclave for the same inconsistent measurement reason.

Host Enclave. Creating a host enclave is also the same as the current enclave. To enable mapping plugin enclaves into its address space, the host enclave must finish its initialization using EINIT, because since EINIT the enclaves can start to attest one another. To reuse a readily initialized plugin enclave, a host enclave uses EMAP to map an immutable enclave region into its private address space, and requires the OS to update the corresponding page table entries (PTEs). EAUG and EMAP can be used commutatively, and CPU will check whether a particular virtual address has been occupied, therefore rejecting conflicted operations. Likewise, EREMOVE and EUNMAP can be used commutatively to tear down a host enclave. As with current SGX design, all private EPCs must be removed and all plugin enclaves must be unmapped to finally remove the SECS page of a host enclave.

The lifecycle of PIE enclaves are depicted in Figure 6. Other lifecycles such as thread management, interrupt handling (asynchronous exit), and context switches (ecalls/ocalls) are all identical to current SGX design.

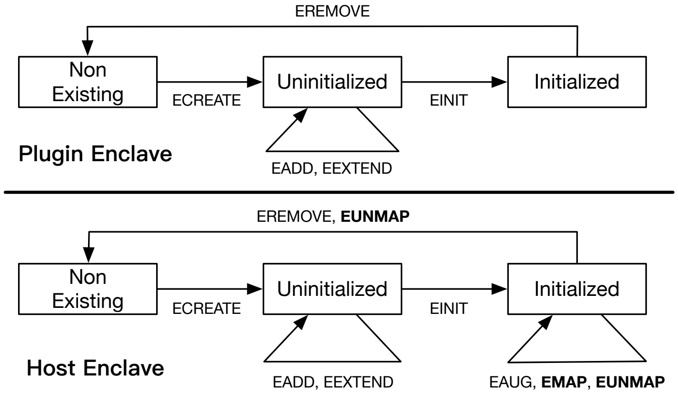


Fig. 6: Lifecycles of plugin enclave and host enclave in PIE.

F. Hardware/Software Update Summary

Hardware Modification. PIE's design includes two new instructions (EMAP and EUNMAP), a new page type (PT_SREG), and extends the SECS field to maintain the mapping relationship. EPC access control validation and page eviction mechanism should be extended to support PIE. The copy-on-write mechanism for shared EPC reuses SGX2 EACCEPTCOPY. PIE does not change memory encryption engine (MEE). Because all EPC pages are encrypted by MEE using a global key, plugin enclaves can be directly accessed by permissible host enclaves.

It is known that the majority of Intel SGX was implemented using CPU microcode [40] and SDM describes that the CPU microcode can be updated [29]. Hence, PIE's extension can be applied using SGX microcode update without modifying Intel CPU hardware logics.

Building a PIE Enclave. PIE is fully compatible with existing SGX toolchain, where a developer writes Enclave Definition Functions (EDL) to declare o/ecalls, applies the edger8r tool to generate glue code for application/enclave code, and finally signs an enclave report within Signature Structure (SIGSTRUCT). PIE has a slight modification to the final step: the developer should enumerate a list of hashes of valid plugin enclaves in a manifest, in order for the host enclave to check against them via local attestation (see below).

PIE Remote/Local Attestation. PIE reuses existing SGX attestation infrastructure. Intel provides a vendor-defined enclave named Launched Enclave (LE) that assists local attestation. We propose modifying this LE for efficient attestation of PIE serverless, as depicted in Figure 7. LE provisions an identity management service which allows a host enclave to quickly identify different versions of plugin enclaves. As a result, users only need one remote attestation (RA) instead of multiple RAs, and the remainder consists of multiple local attestation (LA), which is extremely efficient (merely 0.8ms on our testbed). Multi-version allows PIE address layout re-randomization (see § VII) and can minimize VA conflicts of plugin enclaves.

V. Evaluation Methodology

According to our microbenchmark (§ III), we found that code-intensive applications do not benefit from SGX2's new feature (i.e., EAUG-based dynamic loading). Since released SGX1-capable Intel machines have higher frequencies, we evaluate PIE using a cloud bare-metal server with 8-core Intel Xeon E3-1270 CPU at 3.80GHz and 64GB DDR4 running CentOS 7.6 with kernel 4.1.0. This is reasonable because cloud server are normally equipped with high-end processors.

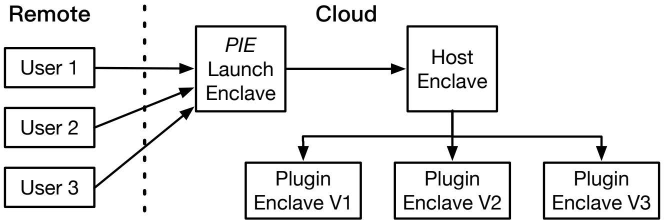


Fig. 7: PIE modifies the Launched Enclave (LE) service for a host enclave to quickly attest different versions of plugin enclaves. A remote user only needs to attest the host enclave once, and the host enclave will attest the needed plugin enclaves for the rest of the attestation chain. The arrows denote the attestation directions. Multi-version plugin enclaves allows for minimizing virtual address conflicts when EMAP'ing.

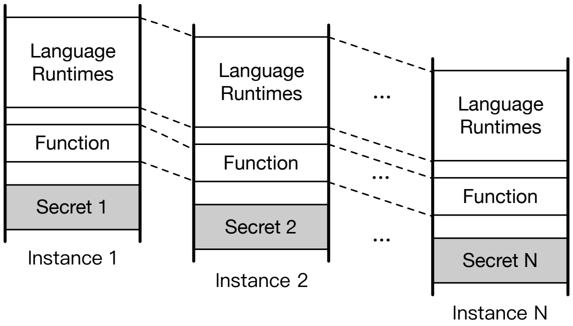
TABLE IV: Emulation cycles of PIE instructions.

Instruction	Cycles	Semantics
EMAP	9K	Add Plugin EID into Host's SECS
EUNMAP	9K	Remove Plugin EID from Host's SECS

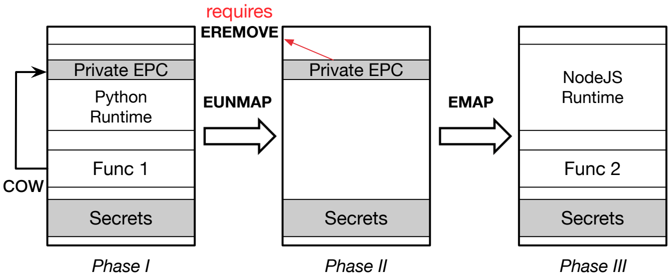
Instruction Emulation. We extend Graphene-SGX to emulate PIE's new instructions for plugin enclave (un)mappings. As PIE does use SGX2 instructions (e.g., EACCEPTCOPY), we use the measured cycles from SGX2 machines and add this latency to PIE-based copy-on-write mechanism. Because EMAP/EUNMAP only updates the SECS metadata, we use the latency of SGX2 EMODPE instruction, which is the only user-level instruction that also updates the metadata (EPCM). Table IV presents the emulated latency of PIE.

Performance Model. We use POSIX multi-threading model inside one real-world SGX enclave to emulate multiple host enclaves sharing plugin enclaves. To emulate EMAP process, we use the *ioctl()* system call to instruct the SGX driver to map a memory region into the enclave address space by updating the corresponding page table entries. To emulate the COW mechanism, we set the permissions of the shared pages to be read only. When the host enclave writes a shared page for the first time, our modified driver will add the copy-on-write (COW) latency measured from EAUG and EACCEPTCOPY (20K cycles in total). To emulate EUNMAP process, the host enclave needs to explicitly zero private pages caused by the runtime COW. We add the latency measured from EREMOVE (4.5K cycles) for each page zeroing. PIE's access control requires additional EID checks when loading an evicted EPC page, in which EID validation happens on each TLB miss. We use the Linux Perf Tool based on Intel Performance Monitoring Unit (PMU) to collect the end-to-end TLB miss occurrences (dTLB miss and iTLB miss included), and add this EID checking overhead (4∼8 cycles per TLB miss) accordingly.

Host/Plugin Partitioning. In our evaluation, runtimes like Python and NodeJS, official packages like Tensorflow and OpenSSL, and the open-source serverless functions are mapped to host-enclaves; private ML models, and user data (i.e., secrets) are mapped to plugin-enclaves. Because the real-world serverless applications (Table I) we benchmarked did not use private shared objects, we mapped all used libraries into shareable plugin enclaves. In principle, all the data and code deemed non-secret can be mapped to plugin enclaves.



(a) **Using PIE EMAP to reuse the common states when serving autoscaling.** To create a new function instance, the cloud platform only needs to create a new host enclave, and EMAPs the common states such as language runtime and function.



(b) **Using PIE instructions to remap functions to achieve *in-situ* processing in a function chain.** In phase I, any write to shared EPC triggers copy-on-write and inserts a private EPC. To remap another function, the host enclave should first EUNMAP the old function and its runtime, and EREMOVE private EPCs (otherwise the VA range will be conflicted), as shown in phase II. In phase III, the host enclave EMAPs a new function to proceed the service.

Fig. 8: PIE-based serverless optimizations.

VI. Evaluation

This section evaluates the performance improvement of applying PIE to the serverless workloads we studied in § III. We compare three scenarios: (1) **SGX-based cold start**: a software-optimized environment (using hardware/software measurement and template-based techniques proposed in § III-B) where each enclave is created on demand upon a new request. (2) **SGX-based warm start**: a "smart" environment which speculatively pre-warms a number of enclaves ready to serve concurrent requests within a capacity (30 instances on our testbed); a software reset must be performed between invocations for privacy reason. (3) **PIE-based cold start**: a PIE-optimized environment (as demonstrated in Figure 8a and Figure 8b) where a number of plugin enclaves are created in advance, but host enclaves for serverless functions are created on demand which is similar to (1). We evaluate these scenarios in terms of function startup latency (§ VI-A), autoscaling (§ VI-B) and function chaining (§ VI-C). We also compare EPC eviction counts because PIE's secure sharing reduces EPC duplication (§ VI-D).

A. Single Function Statup Improvement

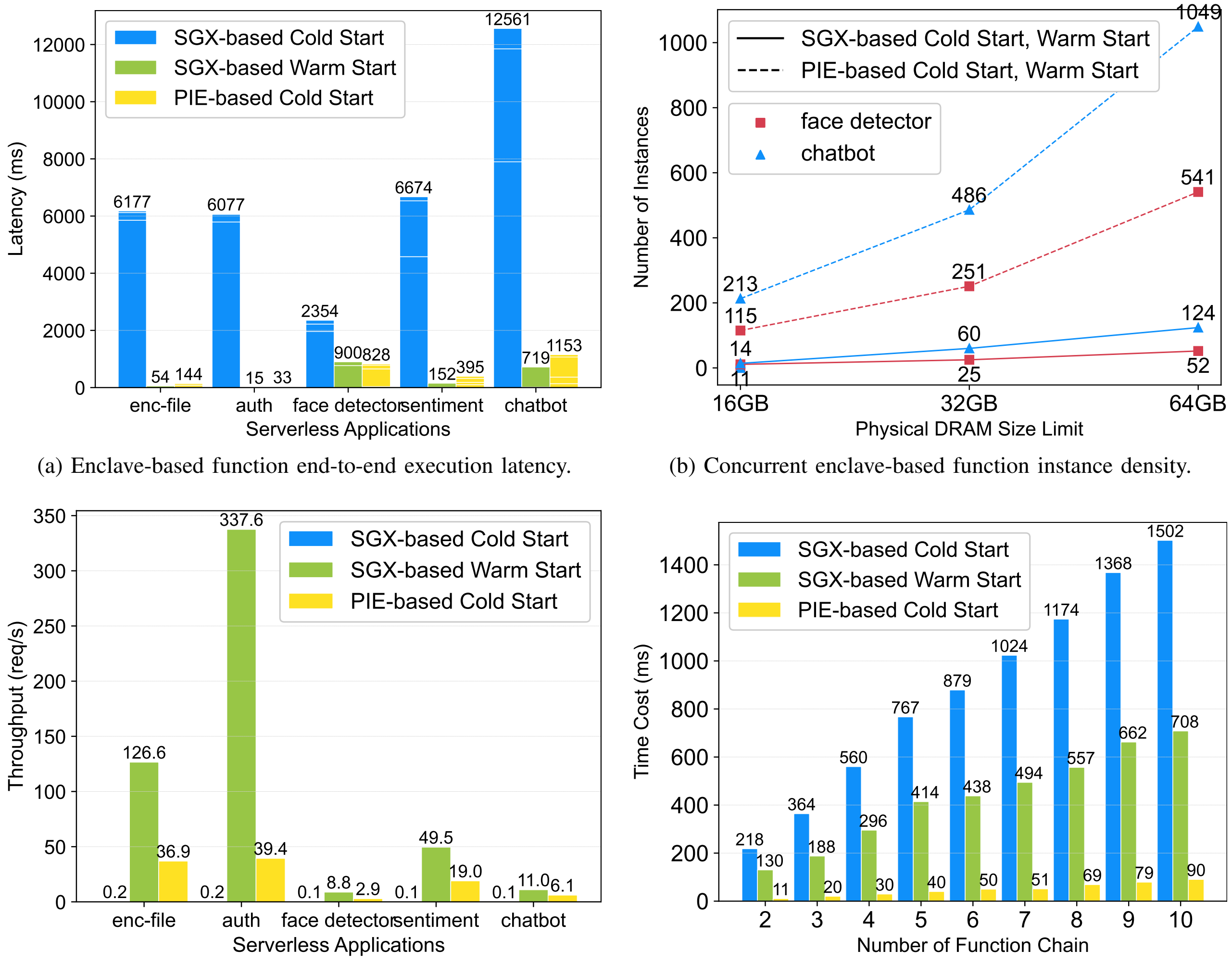
As depicted in Figure 9a, SGX-based warm start achieves the fastest startup as it preserves enough enclaves instances ready to use. Utilizing PIE for on-demand startup (which respects the spirit of serverless computing) incurs no more than 200ms on average to the response latency, but saves 28GB memory compared with SGX-based warm start. The only exception is the face-detector application, which requires around 122MB size of EPC heap for each request, and incurs 618ms latency in total. In terms of startup latency (execution time excluded), PIE-based cold start is 3.2× to 319.2× faster than SGX-based cold start, as PIE avoids code page initialization and measurement generation. For end-to-end latency, PIE-based cold start is 3.0× to 196.0× faster than SGX-based cold start, while it only preserves around 2GB memory, compared with 60GB memory of SGX-based warm start.

B. Autoscaling Improvement

As discussed in § III, current SGX design cannot well support enclave autoscaling, because concurrent enclave startup incurs intensive EPC evictions. As shown in Figure 9c, the throughputs of cold start is less than 0.22 requests/second, and its latency is more than 71s on average, which is impractical at all. PIE-based cold start achieves 94.75% to 99.5% reduction in latency, and 19.4x to 179.2x increase in throughputs compared with SGX-based cold boot. Still, we observe a response slowdown (from 781ms to 2625ms, 3.4× slowdown) for the face-detector application in PIE-based cold start, due to EPC contentions of concurrent 122MB heap allocations, whereas SGX-based warm start is not affected. For heap-intensive enclave functions, we suggest the serverless platforms to leverage **PIE-based warm start**, which pre-warms a number of host enclaves ready to serve. PIE-based warm start saves more memory resources than SGX-based warm start. As shown in Figure 9b, PIE-based serverless allows much higher enclave function density (10× than current SGX). It is the platform's choice to trade off between resource usage and quality-of-service (service latency and throughputs).

C. Function Chaining Improvement

We use a simple Python resize function and a real-world personal photo (10MB) as secret data to test the data transfer cost while increasing the length of the enclave function chain. Figure 9d shows data transfer cost between functions. Both SGX-based cold start and SGX-based warm start need to



(a) Enclave-based function end-to-end execution latency.

(b) Concurrent enclave-based function instance density.

(c) Enclave-based function autoscaling throughputs.

(d) Data transfer cost in an enclave-function chain.

Fig. 9: Performance comparison amongst SGX-based cold-start, SGX-based warm-start and PIE-based cold-start serverless.

copy secret data across enclave boundaries. SGX-based warm start saves the cost of enclave heap allocation because of pre-allocation, and is 2.1x faster than SGX-based cold start. PIE-based cold start outperforms the other two (16.6-20.7× over SGX-based cold start and 7.8-12.3× over SGX-based warm start) because its remapping-based *in-situ* processing avoids redundant copy/marshalling and removes additional encryption/decryption overhead. Since the functions are all written in Python, PIE only needs to EUNMAP function logic and the corresponding package plugin enclaves. On each remapping, PIE EREMOVEs 2,819 copy-on-write private pages and incurs 8.47ms on average. Note that PIE-based remapping is more suitable for data-transfer-intensive workloads. For small message passing (e.g., ∼100KB), *in situ* processing may not be effective.

D. EPC Eviction Reduction

SGX is known to suffer from EPC eviction overhead which significantly slows down enclave performance [24], [32]. Since PIE EMAP reuses content-ready EPC pages, it avoids host enclaves to allocate more pages which must occupy the physical EPC pool. Allocating EPC from the pool is very likely to trigger an EPC eviction process. Table V shows that both SGX-based warm start and PIE-based cold start incurs no more than 1.0% of EPC evictions compared with SGX-based cold start. Again, in the case of face-detector, SGX-based warm start and PIE-based cold start incur relatively higher evictions (10%∼11%), because of software reset or heap allocation activities. Still, due to the limited size of physical EPC, intensive EPC usage still incurs very high overhead. It turns out that PIE design mitigates this.

TABLE V: Counting EPC evictions during autoscaling.

Application	enc-file	auth	face-detector	sentiment	chatbot
Cold Start	42.9M	43.5M	47.8M	107.2M	166.9M
Warm Start	78.0K	78.0K	5.0M	468K	1.2M
PIE-based Start	98.6K	98.6K	5.3M	468K	1.7M

VII. Security Analysis

This section assesses the security implications of PIE, with respect to its architectural modification for enclave sharing and mapping semantics.

Sharing Plugin Enclaves: PIE's plugin enclaves contain non-sensitive environments such as language runtimes and frameworks, and therefore are safe to be shared amongst different functions. The CPU guarantees the correctness of these shared states, proves their identities via CPU-generated measurements, and allows them to be attested anytime when they are mapped into the host enclaves.

Attacking Plugin Enclaves' Measurement: A plugin enclave must finalize its measurement to be shared via `EMAP`. Once being initialized, both measurement and its content of a plugin enclave are locked down. CPU disallows SGX2 instructions such as `EAUG` to change its content. Any writes from host enclaves will trigger hardware-enforced copy-on-write that updates private EPCs of the host enclave. Moreover, EPC pages reclaim such as `EREMOVE` on a plugin enclave always terminates the possibility of further sharing. This strict model establishes the trustworthiness of using a plugin enclave.

Stale Mapping After `EUNMAP`: A host enclave is still able to access an `EUNMAP`'ed plugin enclave if its TLB has not been flushed. For a multi-threaded host enclave, it can either use an in-enclave flag to make sure all threads have reached a quiescent point before `EUNMAP`'ing a particular plugin [41], or define `EUNMAP` to automatically trigger an enclave exit on all CPU cores. To optimize the latter, we may use cache-coherence-alike mechanism to only shootdown TLBs of the related CPU cores of the same host enclave EID.

Malicious Mapping From OS: Even if the OS configures wrong page tables for a host enclave, an enclave cannot access the shared EPCs that are not explicitly `EMAP`'ed. Private EPCs are already forbidden to be shared in current SGX hardware.

Malicious Plugin Enclaves: As PIE's threat model assumes all in-enclave code is trusted, a PIE developer should only include trusted plugins in the manifest. A host enclave must verify their measurements before `EMAP`ing plugin enclaves, hence excluding malicious plugins during runtime.

Address Space Layout Randomization (ASLR): Because PIE's design allows multiple host enclaves to reuse the same plugin enclave, a nefarious attacker may combine brute force and memory disclosure to bypass ASLR. In fact, supporting ASLR is challenging for systems that apply memory sharing for better performance. A practical mitigation approach is batching: e.g., applying ASLR for every 1,000 enclave creations, instead of every enclave. This approach is effective since real attacks usually require thousands of test-and-trials with the same memory layout [42]. Techniques like runtime ASLR [43] can also be applied to further raise the bar of successful attacks, which are orthogonal to our work. The randomization frequency is adjustable for PIE developers to make a security/performance tradeoff.

Side-channel Analysis: In contrast to SGX share-nothing model, PIE brings page-sharing side-channel to its neighbors. Consider two enclaves using the same library. In SGX, each enclave has its own copy of the library. In PIE, two host enclaves share one library plugin enclave with copy-on-write. As a result, the host enclave can learn: (1) how the library memory pages are mapped, and (2) whether a library page of the other host enclave is in memory or not (through a timing channel). Both information can also been seen by a malicious OS, and PIE further exposes them to the neighboring enclaves. Prior mitigations against the malicious OS [44], [45] can thwart attacks from both OS and neighboring enclaves.

VIII. Discussion

A. Compared with other solutions

Microkernel-like Sharing. Conclave [46] proposes using multiple server enclaves to be shared between application enclaves. However, the low-latency property required by serverless computing is hard to achieve due to the unshared nature of multiple enclave address spaces. Secret data must be re-encrypted across enclave boundaries using an SSL-like secure channel, which inevitably incurs high overhead, especially for data-intensive workloads (as evaluated in Figure 3c). Even worse, this solution cannot deal with a heavyweight language runtimes (LR) shared across many function enclaves, where each function enclave has to contain an independent LR.

Unikernel-like Sharing. Occlum [34] allows efficient multi-tasking within a single enclave address space, by sharing a library OS between many software-based isolated tasks. Occlum enables fast *spawn()* system call, which is suitable for serverless autoscaling. We deem Occlum as a software alternative to PIE. The major difference is that Occlum's in-enclave isolation is guaranteed by software, which requires comprehensive code instrumentation and complicated integrity checks (e.g., control-flow integrity), while PIE only puts trust in hardware to isolate different host enclaves. It is rather hard to prove Occlum's software-based isolation is adequate given prevailing software memory bugs [47].

Nested-library Sharing. Nested Enclave [35] introduces hardware-based hierarchical isolation within an enclave. Nested Enclave places shared libraries in a shareable outer enclave, while running each user logic in an independent inner enclave. While being shared, the outer cannot access the inner. Both PIE and Nested Enclave can benefit serverless autoscaling with rapid instantiation. PIE differs from Nested Enclave in that PIE provides N:M mappings between host and plugin enclaves, whereas Nested Enclave only supports N:1 mapping between inner and outer enclave. Nested Enclave may not be a good fit for serverless computing in the following aspects: (1) It is impossible to share interpreted LR (e.g., NodeJS, Python) in the outer enclave, because these runtimes must access user scripts in the inner enclaves. (2) Nested Enclave replaces library calls with enclave calls, which requires code modification and incurs runtime context-switch overhead (6K∼15K cycles). By contrast, PIE allows a host enclave to invoke a plugin enclave via fast function calls (60 cycles). Nevertheless, Nested Enclave's asymmetric access

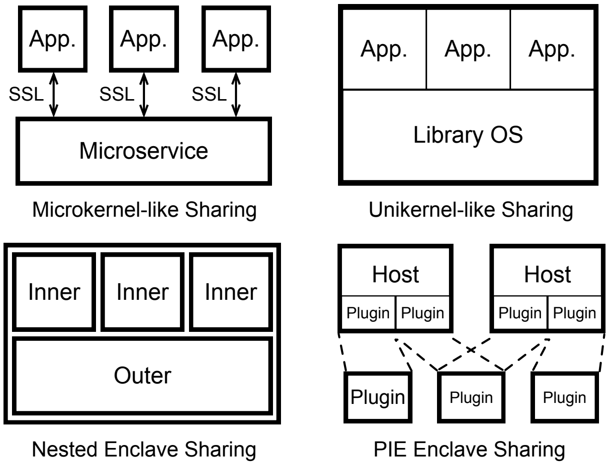


Fig. 10: **Comparison between PIE and other options.** The darker border represents hardware-based enclave isolation.

model isolates library bugs from user logics, whereas PIE remains the same monolithic model as current SGX design.

B. Optimizing other workloads

PIE can be used to optimize more enclave applications other than serverless workloads. For enclave programs written in high-level languages [48], PIE reduces both the creation time and memory footprints when the user launches multiple program instances. Moreover, PIE enables lightweight POSIX *fork()* system call via its copy-on-write mechanism, whereas in current SGX design, the enclave *fork()* has to copy the whole in-enclave content [21]. Another opportunity is that PIE can boost privacy-preserving AI training because these workloads require massive data communication between each executors [9]. The secret data transfer between enclave executors can benefit from PIE's remapping technique.

C. Compatible with existing serverless platforms

Existing serverless platforms such as AWS Lambda [1] require application developers to maintain the function dependencies and upload a self-contained image. In PIE enclave-based serverless, the developers are encouraged to use public images of large size released by cloud vendors, and leverage enclave local attestation to verify the image identity. Running serverless applications in PIE does not require any rewriting of the functions, but to include the hashes of public enclave images in the manifest of the host enclave, which respects the spirit of serverless—offloading most labor work to the cloud.

IX. Related Work

SGX Enclave Optimization and Enhancement. A line of previous works have focused on optimizing SGX performance. VAULT [32] introduces a variable arity tree to protect the entire physical memory. InvisiPage [49] proposes using page-level protection instead of cache-block level to expand EPC size. Eleos [24] provides software-based MEE, while CoSMIX [50] leverages compiler-based MEE. All these efforts aim to reduce EPC eviction overhead. Asynchronous message queues are proposed in HotCalls [23] and SCONE [22] to avoid e/ocall context switches. All the above optimizations are orthogonal to PIE, and can be combined with PIE to make enclave applications more efficient and practical. Nested Enclave [35] proposes a single outer enclave to be shared by multiple inner enclaves, while PIE allows more enclaves to be shared and enables cheap function calls. A more comprehensive comparisons is explained in § VIII-A.

Enclave-based Serverless Frameworks. Recently, some system works have utilized trusted hardware to protect serverless applications. Se-Lambda [18] leverages a WebAssembly sandboxed environment as a two-way function for serverless functions. S-FaaS [19] combines hardware transaction (namely Intel TSX) and SGX for trusted serverless resource account. T-FaaS [20] ports JavaScript engines into SGX to build a secure serverless platform. Clemmys [10] batches SGX2 EAUG operations for fast creation of a large-heap serverless enclave. Clemmys' technique cannot optimize the latency introduced by large-code enclave initialization and measurement, which PIE solves by securely reusing immutable plugin enclaves.

Serverless Startup Latency Optimization. There is an increasing attention to the latency issue of serverless computing in research academia. Shahrad et al. [51] investigate architectural implications of serverless applications on modern processors. Liang Wang et al. [52] measure the cold-start latency on commercial serverless platforms. To reduce the function startup cost, SAND [5] exploits fine-grained sandboxing and a high-locality message bus, while SOCK [6] suggests using lean containers. Catalyzer [7] achieves sub-millisecond functions startup with copy-on-write and in-memory snapshot sharing. FAASM [9] leverages shared memory to avoid expensive data migration between functions. With PIE extension, the optimization techniques of Catalyzer and FAASM can be directly and securely applied to enclave-protected functions.

X. Conclusion

Intel SGX is designed for confidential cloud computing. Our benchmarks show that existing enclave hardware cannot retain low latency when serving serverless workloads. PIE extends Intel SGX with shareable plugin enclaves to reuse non-secret heavyweight states, and removes data transfer bottlenecks by remapping enclave functions. Benefited from PIE, enclave functions can reduce 94.74-99.57% startup latency, 19-179× speedup in autoscaling throughputs, 16.6-20.7× improvement in secret data transfer, and achieve 10× function density.

Acknowledgments

We sincerely thank the anonymous reviewers and our shepherd for insightful and constructive comments. This work is supported in part by Key-Area Research and Development Program of Guangdong Province (NO.2020B010164003), China National Natural Science Foundation (No. 61972244, U19A2060, 61925206, 61732010). Yubin Xia is the corresponding author.

References

[1] "Aws lambda." https://aws.amazon.com/lambda/.

[2] "Azure functions." https://azure.microsoft.com/en-us/services/functions/.

[3] "Google cloud functions." https://cloud.google.com/functions/.

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

[5] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: towards high-performance serverless computing," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.*, pp. 923–935, 2018.

[6] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "SOCK: rapid task provisioning with serverless-optimized containers," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.*, pp. 57–70, 2018.

[7] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020*, pp. 467–481, ACM, 2020.

[8] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, "SEUSS: skip redundant paths to make serverless fast," in *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, pp. 32:1–32:15, ACM, 2020.

[9] S. Shillaker and P. R. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, pp. 419–433, USENIX Association, 2020.

[10] B. Trach, O. Oleksenko, F. Gregor, P. Bhatotia, and C. Fetzer, "Clemmys: towards secure remote execution in faas," in *Proceedings of the 12th ACM International Conference on Systems and Storage, SYSTOR 2019, Haifa, Israel, June 3-5, 2019*, pp. 44–54, 2019.

[11] "Serverless examples." https://github.com/aws-samples.

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

[13] "Api gateway custom authorizer function + auth0." https://github.com/serverless/examples/tree/master/aws-python-auth0-custom-authorizers-api.

[14] "Serverless alexa skill." https://github.com/serverless/examples/tree/master/aws-node-alexa-skill.

[15] "Serverless aws rekognition finpics." https://github.com/rgfindl/finpics.

[16] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*, p. 11, ACM, 2013.

[17] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. V. Rozas, "Intel® software guard extensions (intel® SGX) support for dynamic memory management inside an enclave," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, HASP@ICSA 2016, Seoul, Republic of Korea, June 18, 2016*, pp. 10:1–10:9, ACM, 2016.

[18] W. Qiang, Z. Dong, and H. Jin, "Se-lambda: Securing privacy-sensitive serverless applications using SGX enclave," in *Security and Privacy in Communication Networks - 14th International Conference, SecureComm 2018, Singapore, August 8-10, 2018, Proceedings, Part I*, pp. 451–470, 2018.

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

[20] S. Brenner and R. Kapitza, "Trust more, serverless," in *Proceedings of the 12th ACM International Conference on Systems and Storage, SYSTOR 2019, Haifa, Israel, June 3-5, 2019*, pp. 33–43, 2019.

[21] C. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library OS for unmodified applications on SGX," in *2017 USENIX Annual Technical Conference, USENIX ATC 2017, Santa Clara, CA, USA, July 12-14, 2017.*, pp. 645–658, 2017.

[22] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, D. Goltzsche, D. M. Eyers, R. Kapitza, P. R. Pietzuch, and C. Fetzer, "SCONE: secure linux containers with intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pp. 689–703, USENIX Association, 2016.

[23] O. Weisse, V. Bertacco, and T. M. Austin, "Regaining lost cycles with hotcalls: A fast interface for SGX secure enclaves," in *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pp. 81–93, 2017.

[24] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: Exitless OS services for SGX enclaves," in *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, pp. 238–253, 2017.

[25] "Ibm cloud functions." https://www.ibm.com/cloud/functions.

[26] "Apache OpenWhisk is a serverless, open source cloud platform." https://openwhisk.apache.org/.

[27] "Fn Project Open Source. Container-native. Serverless platform.." https://fnproject.io/.

[28] "gVisor is an application kernel for containers that provides efficient defense-in-depth anywhere.." https://gvisor.dev/.

[29] Intel, "Intel® 64 and IA-32 Architectures Software Developer Manuals." https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

[30] dashbird, "Serverless Most Popular Programming Languages." https://dashbird.io/blog/serverless-most-popular-programming-languages/. Access time: 10/11/2020.

[31] F. M. Corey, "The State of AWS Lambda Supported Languages and Runtimes." https://www.serverless.com/blog/aws-lambda-supported-languages-and-runtimes/. Access time: 10/11/2020.

[32] M. Taassori, A. Shafiee, and R. Balasubramonian, "VAULT: reducing paging overheads in SGX with efficient integrity verification structures," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, pp. 665–678, ACM, 2018.

[33] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, pp. 533–549, 2016.

[34] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and efficient multitasking inside a single enclave of intel SGX," in *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020* (J. R. Larus, L. Ceze, and K. Strauss, eds.), pp. 955–970, ACM, 2020.

[35] J. Park, N. Kang, T. Kim, Y. Kwon, and J. Huh, "Nested enclave: Supporting fine-grained hierarchical isolation with SGX," in *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*, pp. 776–789, IEEE, 2020.

[36] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pp. 1–19, IEEE, 2019.

[37] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pp. 991–1008, USENIX Association, 2018.

[38] J. V. Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "LVI: hijacking transient execution through microarchitectural load value injection," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pp. 54–72, IEEE, 2020.

[39] R. Leslie-Hurd, D. Caspi, and M. Fernandez, "Verifying linearizability of intel® software guard extensions," in *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, vol. 9207 of *Lecture Notes in Computer Science*, pp. 144–160, Springer, 2015.

[40] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 86, 2016.

[41] J. Gu, Z. Hua, Y. Xia, H. Chen, B. Zang, H. Guan, and J. Li, "Secure live migration of SGX enclaves on untrusted cloud," in *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*, pp. 225–236, IEEE Computer Society, 2017.

[42] A. Bittau, A. Belay, A. J. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pp. 227–242, IEEE Computer Society, 2014.

[43] K. Lu, W. Lee, S. Nürnberger, and M. Backes, "How to make aslr win the clone wars: Runtime re-randomization.," in *NDSS*, 2016.

[44] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*, pp. 53–70, USENIX Association, 2016.

[45] M. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: eradicating controlled-channel attacks against enclave programs," in *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, The Internet Society, 2017.

[46] S. Herwig, C. Garman, and D. Levin, "Achieving keyless cdns with conclaves," in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pp. 735–751, USENIX Association, 2020.

[47] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pp. 48–62, IEEE Computer Society, 2013.

[48] F. Shaon, M. Kantarcioglu, Z. Lin, and L. Khan, "Sgx-bigmatrix: A practical encrypted data analytic framework with trusted processors," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pp. 1211–1228, ACM, 2017.

[49] S. Aga and S. Narayanasamy, "Invisipage: oblivious demand paging for secure enclaves," in *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, pp. 372–384, ACM, 2019.

[50] M. Orenbach, Y. Michalevsky, C. Fetzer, and M. Silberstein, "Cosmix: A compiler-based system for secure memory instrumentation and execution in enclaves," in *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, pp. 555–570, 2019.

[51] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of function-as-a-service computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, pp. 1063–1075, ACM, 2019.

[52] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018.*, pp. 133–146, 2018.