Reusable Enclaves for Confidential Serverless Computing

Shixuan Zhao, *The Ohio State University;* Pinshen Xu, *Southern University of Science and Technology;* Guoxing Chen, *Shanghai Jiao Tong University;* Mengya Zhang, *The Ohio State University;* Yinqian Zhang, *Southern University of Science and Technology;* Zhiqiang Lin, *The Ohio State University*

https://www.usenix.org/conference/usenixsecurity23/presentation/zhao-shixuan

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

Reusable Enclaves for Confidential Serverless Computing

Shixuan Zhao[1], Pinshen Xu[2], Guoxing Chen[3], Mengya Zhang[1], Yinqian Zhang[2], Zhiqiang Lin[1]

[1]*The Ohio State University*
[2]*Southern University of Science and Technology*
[3]*Shanghai Jiaotong University*

Abstract

The recent development of Trusted Execution Environment has brought unprecedented opportunities for confidential computing within cloud-based systems. Among various popular cloud business models, serverless computing has gained dominance since its emergence, leading to a high demand for confidential serverless computing services based on trusted enclaves. However, the issue of cold start overhead significantly hinders its performance, as new enclaves need to be created to ensure a clean and verifiable execution environment. In this paper, we propose a novel approach for constructing reusable enclaves that enable rapid enclave reset and robust security with three key enabling techniques: *enclave snapshot and rewinding*, *nested attestation*, and *multi-layer intra-enclave compartmentalisation*. We have built a prototype system for confidential serverless computing, integrating OpenWhisk and a WebAssembly runtime, which significantly reduces the cold start overhead in an end-to-end serverless setting while imposing a reasonable performance impact on standard execution.

1 Introduction

Recently, confidential cloud computing has garnered substantial interest due to the rapid development of Trusted Execution Environment (TEE), an emerging hardware feature offered by most main-stream general-purpose CPUs to safeguard software applications from compromised or malicious system software. Prominent TEEs include Intel's SGX and TDX, AMD's SEV, and Arm's CCA. Confidential cloud computing leverages TEEs to shield cloud tenants' sensitive code and data from cloud providers, offering cloud tenants additional security assurance. Major cloud platforms such as Azure [1], Google Cloud [2], and Aliyun [3] have all introduced confidential cloud services.

Among various cloud business models, serverless computing has gained significant popularity since its inception, which entails automated provisioning and scaling of computing resources on demand, without manual intervention from cloud customers. The name "serverless" refers to an illusion of computing without building complex cloud infrastructures. Function as a Service (FaaS) [4] is a form of serverless computing, allowing customers to execute code in response to requests with the ability to rapidly scale up and down. The concept of constructing confidential serverless computing by *protecting the backend executors within TEEs* has been investigated in several prior studies [5–7] and has even been commercialised [8]. In such settings, as illustrated in Figure 1, a *frontend gateway* handles the user requests and forwards them to a set of *backend executors* that carry out the actual user function execution. Executions of users' functions within enclaves ensure the confidentiality and integrity of the process.

However, the *cold start problem* poses a significant challenge for serverless computing. As functions must quickly scale up to accommodate large amounts of client requests, code and data must be loaded into the memory and execute the function with minimal delays [9]. The cold start of a function involves time-consuming steps such as creating and instantiating a new runtime environment required to launch the user function. In confidential serverless, since the runtime environment is protected within an enclave, the process inevitably requires creating a new enclave, which is more costly. As to be shown in section 7, creating an SGX enclave with an industry median of 170 MiB of memory takes 2482 ms. In contrast, recent studies have demonstrated that the industry median execution time for a serverless workload is within 1000 ms [10] [11]. The cold start time is more than twice the duration of an average workload.

While techniques have been developed for traditional serverless scenarios by keeping the execution environment warm and reusing it instead of reinitialising [9] [12], they are not applicable to confidential serverless. This is because a malicious function workload could exploit vulnerabilities in the function runtimes (e.g., CVE-2023-26489 [13]) to compromise the enclave environment, which would also expose other users' workloads if the enclave is reused. As discussed in the Conclave's talk [14], for higher security

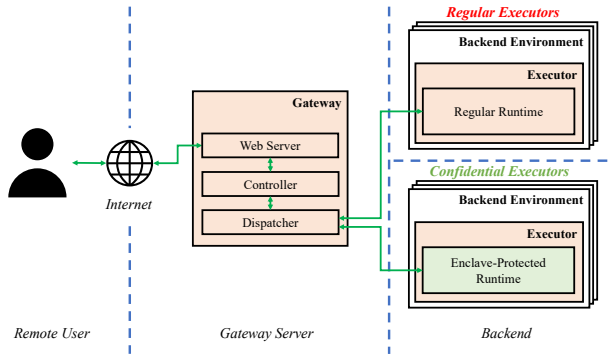


Figure 1: A typical setup of regular and confidential serverless computing.

assurance, *an enclave must be discarded after every execution* in a confidential serverless setting.

In this paper, we propose *reusable enclaves* to facilitate enclave (and hence the function executor) reuses in serverless settings, enabling the cold start problem to be addressed within the realm of confidential serverless computing. Specifically, the concept is enabled by three key techniques: *enclave snapshot and rewinding*, *nested attestation*, and *multi-layer intra-enclave compartmentalisation*. First, enclave snapshot and rewinding allows for saving a known good enclave state as a snapshot and rewinding the enclave to that state later. This technique enables the rapid rest of a serverless executor enclave to its initial state after executing a workload, ensuring the security of the subsequent workload without resorting to costly enclave relaunching. Second, nested attestation permits the attestation of the runtime environment following reset. Last but not least, multi-layer intra-enclave compartmentalisation (MLIEC) employs an extended software fault isolation technique to segregate the enclave into multiple security layers, preventing code in less-secure layers from accessing more-secure ones, and thereby ensuring the security of secure layers even after executing malicious serverless functions.

To showcase the concept, we have developed a comprehensive prototype ranging from the toolchain to a fully functional system. Specifically, we created a compiler-assisted toolchain based on LLVM 11.0 to implement the MLIEC techniques, achieving complete Intel SGX SDK instrumentation. Subsequently, we employed this toolchain to integrate the entire reusable enclave mechanism into a prototype confidential serverless system based on OpenWhisk and WebAssembly Micro Runtime with Intel SGX. We conducted performance evaluations of cold start latency, execution overheads due to instrumentation, and end-to-end execution performance of real-world serverless functions. The results indicate that reusable enclave techniques substantially enhance the overall performance of confidential serverless execution by considerably reducing the overhead compared to cold start while imposing a reasonable impact on the workload execution.

In summary, this paper offers the following contributions to confidential serverless computing:

- It proposes a novel solution to the issue of cold start latency in confidential serverless settings, utilizing the innovative concept of reusable enclaves.
- It presents key techniques to realise reusable enclaves, including enclave snapshots and rewinding, nested attestation, and multi-layer intra-enclave compartmentalisation.
- It provides an LLVM-based toolchain for performing automated MLIEC, and a prototype confidential serverless framework employing OpenWhisk, WAMR, and Intel SGX for performance evaluation.

2 Background

In order to solve the cold start problem under the context of confidential computing and serverless clouds, our key insight is to design a fast reset mechanism and protect it using Software Fault Isolation (SFI) techniques. In this section, to facilitate better understanding to our design and implementation, we first introduce the concept of serverless computing and its confidential variant (section 2.1), then the WASM backend executor (section 2.2) and Intel SGX enclaves (section 2.3) used in our demo implementation as the confidential serverless platform, and finally related knowledge of SFI (section 2.4).

2.1 (Confidential) Serverless Computing

Serverless computing is a popular cloud service model in which the cloud provider takes care of the server on behalf of the user and allocates computing resources on demand. Developers could focus more on developing and designing the application code and offload the backend infrastructure management to the cloud provider. Function as a Service (FaaS) is an important computing paradigm within serverless computing, where the applications in the form of functions run only in response to user requests or events. Popular open source FaaS platforms like OpenWhisk [15] use high-level programming language runtimes as the backend executor to run user's functions. Examples of cloud infrastructures that support serverless computing include AWS Lambda [16], Azure Functions [17], Google Cloud Functions [18], and IBM Cloud functions [19]. Efforts like [5–7] have been made to achieve *confidential serverless computing* by protecting the user's execution with TEEs. Confidential serverless has also been commercialised in platforms like Conclave [8]. One major concern about serverless computing is the problem of cold start, *i.e.*, the delay between a request to execute a serverless function and the actual execution of the function caused by setting up an executor environement [9].

2.2 WebAssembly: A Backend Executor

WebAssembly (abbreviated WASM) is a portable, low-level bytecode format developed by W3C Community Group [20]. As a generic bytecode, most high-level programming languages can be compiled into WASM, making its runtime suitable as a backend executor for the serverless computing.

WASM's core is a virtual instruction set architecture (virtual ISA), allowing it to be compatible across different modern hardware and platforms. Various tools have been introduced to compile high-level programming languages to WASM binaries. For example, Emscripten [21] and Binaryen [22] supports C/C++ while LLVM supports more languages. The generated WASM binaries can be executed in WASM runtimes. As a flexible low-level code, efforts have been made to adopt WASM runtimes as backend executors in serverless scenarios [12].

To reduce runtime overhead, ahead-of-time (AoT) compilation could be adopted by compiling WASM code to native code. The interactions between the WASM code (either AoT binary or bytecode) and its underlying environment, such as I/O and system calls, must be handled by a dedicated runtime via defined APIs.

2.3 Intel SGX: Protecting the Executor

Under the confidential serverless context we are targeting, the execution must be protected in a trusted execution environment. As a hardware-protected TEE, Intel SGX well suits this scenario. More specifically, Intel Software Guard Extensions (SGX) [23] is a set of micro-architectural extensions that has been revisioned to SGX2 and is now actively supported targeting the cloud server market, providing a shielded execution environment, called *enclave*, to protect the confidentiality and integrity of the code and data loaded inside against privileged adversaries such as a malicious OS or rogue administrator. A specific range of DRAM, dubbed *Enclave Page Cache* (EPC), is used to hold an enclave's code, data and related data structures. Particularly, the EPC is split into 4 KiB EPC pages, each of which can be assigned exclusively to one enclave and will deny accesses from any software (including the OS) other than the assigned enclave. To use an enclave, one must first perform a heavy launching process that can take a few seconds when the enclave is large as shown in studies [24, 25] and our evaluations in section 7. Following the convention in Intel SGX SDK, an ECall refers to an API called by the untrusted code to execute enclave functions, and an OCall refers to an API used by the enclave code to invoke untrusted code.

2.4 Software Fault Isolation: Sandboxing Using Software

Software Fault Isolation (SFI) is a technique that instruments a piece of code with checks and security enforcement so that any unintentional or malicious behaviours (fault) can be sandboxed and isolated from contaminating other components [26]. A common use of SFI is to create compartments within a single address space so that one compartment cannot access the data in another compartment. This can enable an application to protect certain higher security functions against its regular code when process isolation or enclave isolation is not applicable like within an enclave app itself [27] [28] or within the kernel-space [29]. An example of of such technique is Native Client (NaCl) [30] which utilised the segment registers to divide the address spaces into several isolated sandboxes.

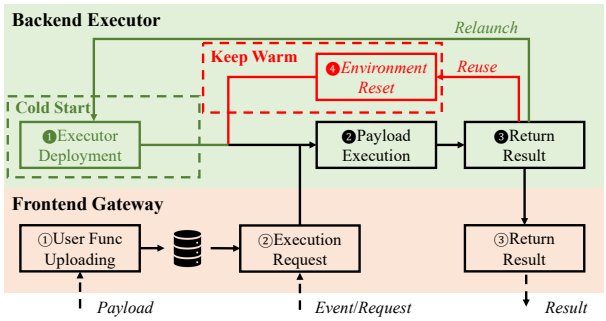


Figure 2: Typical life cycle of a backend executor

3 Overview

3.1 Problem Statement

Cold start latency refers to the time required to launch a serverless function in a completely new execution environment, an overhead that serverless computing aims to minimise. A typical backend executor lifecycle is illustrated in Figure 2. Numerous efforts have been made to reduce this latency by replacing the launching step ❶ with a more efficient one ❹, such as resetting the executor's runtime environment to decrease the code start time [12] or maintaining a warm environment by reusing the same docker [31].

However, these techniques cannot be easily applied to the confidential serverless scenarios, which, ironically, experience even more significant cold start problems due to the substantial overhead of enclave launching in step ❶ [24]. In a serverless architecture, a workload is stored in a database and is only sent to the backend executor upon an execution request/event. This means that after an execution, the next workload to be executed on the same executor might be from *a different user*. Since the runtime in the executor could be vulnerable (containing exploitable bugs), reusing the executor might allow an attacker to deploy a malicious function that exploit the bugs, infecting the runtime and compromising the next user's execution in the same reused executor. Consequently, to eliminate such risks, enclave relaunch is currently ***unavoidable after every execution*** to provide a clean execution environment currently [14].

As demonstrated in recent studies [10, 11] and our own evaluation in section 7, although more than half of serverless workloads finish within 1 second, creating and initialising an SGX enclave incurs an overhead on the order of seconds [24], rendering the cold start overhead challenging to accept. In production solutions like Conclave [14] and previous academic projects [5–7], the approach is to compromise the security by trusting the runtime to be bug-free. Conclave is also considering offering users willing to accept the performance penalty the option to relaunch the enclave [14]. The takeaway is that **there is currently a real-world trade-off between full security guarantee and performance** depending on whether an executor enclave relaunch is performed after each execution. This leads to the research problem: ***Can we design a fast and secure reset to replace the burdensome relaunch*?** In other words, is there a fast and secure step ❹ for confidential serverless computing?

3.2 Threat Model

In the context of confidential serverless computing, our system considers two attack scenarios as well as the possibility of their collusion:

- **Outside Adversaries: OS and Network.** We assume an outside adversary can exploit any exposed interfaces of the runtime as well as OS-level capabilities, including calling any enclave calls with arbitrary parameters, triggering interrupts into the executor enclave, and so on.
- **Inside Adversaries: Malicious Workloads.** We assume a malicious workload can execute any bytecode (or AoT binaries compiled using the provided toolchain) and leverage any bugs within the runtime, such as altering critical parameters (e.g., heap pointers) or even persisting malicious gadgets in memory.
- **Collusion:** The two types of attackers can also collaborate, combining the abilities from both sides, like performing a reset while the malicious workload is executing at a specific critical timing.

We assume the runtime within the enclave to be in good faith but potentially vulnerable. This means that the runtime will correctly execute all the benign instructions given and will not introduce danger or incorrect behaviours unless an adversary deploys a workload to exploit the bugs in it.

We consider DDoS attacks and attacks against the TEE hardware environment (e.g., side-channel attacks [32] and speculative-execution attacks [33] to Intel SGX) to be out of scope.

3.3 Design Goals

To address the problem of cold start in confidential serverless clouds, we propose a secure approach to reuse enclaves for hosting untrusted serverless functions. This design, however, has to meet the following design goals:

- **G1: Serverless architecture.** The system should be tailored for serverless scenarios, capable of executing function workloads from remote users.
- **G2: Isolated execution.** The function workload execution must take place within the enclave, ensuring its confidentiality and integrity, while shielding it from the service provider.
- **G3: Fast and prompt reset.** The execution environment can be securely and promptly reset to its initial state for executing a new workload. A malicious function workload must not contaminate the execution environment.
- **G4: Verifiable reset and deployment.** The environment must provide a mechanism to prove the reset and the identity of the workload to the remote user via an attestation process.
- **G5: Securing the reset and attestation.** The reset and attestation procedures must be adequately safeguarded. The reset must not be interfered by either outside or inside attackers. The attestation must be also protected from the reset module.

3.4 System Overview

Confidential Serverless Platform. Our system is designed for the confidential (**G2**) serverless (**G1**) paradigm, utilizing an enclave-protected high-level programming language runtime as the backend executor. To achieve this, we proposed a design applicable to generic enclave-protected executors of serverless computing in section 4 using software techniques. To demonstrate the technique and evaluate its performance benefits, we also implemented a full-stack system with LLVM, OpenWhisk [15], Intel SGX and WebAssembly Micro Runtime (WAMR) [34] in section 5.

Fast Enclave Reset. One of the main contribution of this paper comes from **G3**. We must provide a fast way to reset the runtime executor enclave. In our system, we designed a mechanism called *enclave snapshot and rewinding*, which allows the serverless system to promptly *reset* and *reuse* an enclave instead of relaunching it, significantly reducing the overhead compared to cold start.

Verifiable Reset and Deployment. Another contribution of this paper arises from **G4**, which requires the enclave to prove the reset is properly conducted and secure the end-to-end execution. We achieved this using an enclave-hosted cryptographic service and a technique we call *nested attestation*. To ensure the trustworthiness of the attestation, we must isolate this module from potential attacks, including those against the reset module.

Reset and Attestation Protection. The third contribution of this paper comes from **G5**, which necessitates securing the reset and attestation. To achieve this, we borrowed the in-enclave SFI idea from SGX-Shield [27] and significantly

expanded it with **unaligned critical function** and **multi-layer compartmentalisation** techniques to enable *multi-layer in-enclave compartmentalisation* (MLIEC). In our system, each thread is assigned a boundary based on its security layer, preventing the thread from accessing or modifying any data of a higher security layer. Transitions between different layers of a thread must also be protected.

MLIEC is the enabling technique of reusable enclaves: smaller in-enclave TCBs can be created and maintained in more secure layers, isolated from the code in the normal execution layer where complex and potentially buggy runtime and user workloads are hosted. With this technique, we can secure the enclave reset and attestation.

4 Design

In this section, we present the detailed design of our approach for performing the fast and secure reset of the enclave. We specifically present our solutions to address the following challenges: (1) how to rewind enclave to its initial state (section 4.1), (2) how to make the reset verifiable (section 4.2), and (3) how to secure the reset procedure (section 4.3). We present our approach in a generic way in this section, making it widely applicable rather than targeting a specific backend executor enclave. A demo implementation applying this approach using OpenWhisk and WAMR on Intel SGX will be provided in section 5.

4.1 Enclave Snapshot and Rewinding

The resetting process should bring the enclave state back to a known good initial state captured at a point after the enclave is launched. The overall snapshot and rewinding idea is straightforward: We first make a copy of the enclave memory as the snapshot, then write the memory back in rewinding. However, capturing the entire enclave's memory would double the memory footprint and slow down the rewinding. The challenge lies in finding a point where the *necessary portion* of the memory included in the snapshot is minimal.

Our first observation is that the binary code itself is write-protected when loaded in most systems. Although dynamic loading is possible, which we discuss in section 4.3.4, we consider the code pages to be unchanged for now. Therefore, the only portions of enclave memory that change are the data (global variables), stack and heap. These portions form the *snapshot* we are about to rewind.

Our second observation is that the heap and stack are not used and remain empty right after the enclave is launched. This observation suggests that we can minimise the snapshot to only the data portion of the enclave if we take a snapshot when the heap and stack are not yet used.

With these two observations, our snapshot-taking procedure involves creating a backup of the global variables' initial values. While tracking the values of an enclave's variables individually may seem nearly impossible, we can use a special linker script to mark their beginning and end since all of the data is stored in `.data` and `.bss` segments. We then create a special buffer in the enclave to store the whole segments as the snapshot when the heap is empty.

The rewinding process involves restoring the stack, heap and the snapshot. The stack and heap, should be empty, so we zero them out. We then copy the snapshot back to the two segments. After these operations, the enclave is reset to its initial state and is ready for reuse. The overhead of this snapshot taking and rewinding process essentially involves memory copying, which can be significantly faster than the enclave launching process.

We designed the snapshot and rewinding mechanism with a buffer and two special function calls that can be wrapped into enclave calls (e.g., ECall in Intel SGX and OpenEnclave on TrustZone) or called in the enclave's internal procedure. The buffer is dedicated to store the snapshot and is not in the scope of enclave reset. The snapshot-taking call must be called when the heap is empty, preferably right after the enclave is launched. However, if an enclave call does not leave anything in the heap but has some valuable information to keep in global variables (e.g., provisioned secret from a remote attestation session), it can be called before the snapshot. The rewinding call can be called at any moment when the enclave is not executing any other enclave calls and needs to be reset.

4.2 Nested Attestation Infrastructure

To fulfill the requirement of a confidential serverless scenario, the reset must be verifiable to every user before they execute a workload. Additionally, user workloads in confidential serverless computing, whether in bytecode or AoT binary form, need integrity checks. The concept of *nested attestation*, which has been available for non-confidential usage [35], can be employed to support this capability. In this subsection, we discuss the idea and the necessary infrastructures to implement nested attestation.

The core idea is to have a software module dubbed *nested attestation module* inside the enclave to perform the attestation. The serverless platform provider can use the enclave hardware's remote attestation to build a secure communication channel with the nested attestation module. A public-private key pair for signing is generated within the nested attestation module, and the private key is never shared to the outside. The private key must also be protected from any other code inside the enclave. This public key is sent to the platform provider for distribution to the end user.

Each time a reset occurs, the user can challenge whether the reset was successful. The enclave must be able to prove the rest took place. For example, a reset counter serving as a nonce can be stored in the reset module which is monotonically increased by one after each successful reset to prevent replay attacks. By creating a signed report with

the nonce counter and the hash of the current data segments using the private signing key, the enclave owner can confirm that a reset was successfully performed by verifying the increased nonce and the matching hash.

Note that other information can also be added to the report, like Intel SGX. The specific design and protocol of nested attestation can vary depending on the enclave. We will present a specific implementation for our system in section 5.

4.3 Multi-Layer Intra-Enclave Compartmentalisation

While the snapshot-and-rewinding technique with nested attestation can enable a fast and verifiable reset of an enclave, ensuring the security of such techniques is not trivial, particularly in a serverless environment where an adversary may try to breach the security by executing a malicious workload. The interpreter for a high-level language runtime might not be bug-free, and a malicious workload might be able to modify the snapshot. The AoT mode can bring native binaries directly into the enclave, posing even higher risks. Moreover, following the principle of modularity and least privilege [36], as the final line of defense, the nested attestation must be able to tell if the reset is performed correctly and if the snapshot is correct (since a malicious OS can reenter the snapshot enclave call) so it should have a even higher security level.

To address this issue, we proposed multi-layer intra-enclave compartmentalisation (MLIEC) using compiler techniques. MLIEC partitions the enclave address space into multiple *security layers*. A thread running with a higher security layer can access the data of a lower security layer but not vice versa. MLIEC must also provide advanced support for serverless scenarios, including preventing re-entrancy attacks to security critical functions and supporting dynamically loaded code.

With MLIEC, we can protect the snapshot and rewinding technique in a higher security layer than the regular enclave code (e.g., the WASM runtime), ensuring that even if the regular enclave environment is compromised, the enclave reset can still be carried out correctly and restore the environment. The nested attestation module can be placed in another layer that is higher than the reset module to ensure the trustworthiness of the attestation result.

We draw inspiration from the compiler-based enforcement of SFI from SGX-Shield [27] and significantly improve upon it. The original SGX-Shield method enabled a single fixed data access boundary that instrumented binary can only access the data with addresses above that boundary, forming a security layer. However, SGX-Shield mainly focused on ASLR. We built our MLIEC by improving it to support:

- **Critical function protection.** While SGX-Shield simply ignored security critical functions that call assembly code to execute dangerous instructions (e.g., `ENCLU` in Intel SGX), we propose a new method to protect these critical functions that require meticulous design.

```
1 // Before:
2   movq  %rax, 0x7(%rdx, %rcx, 3)
3
4 // After: %r15 for boundary, %r14 for offset
5   leaq  0x7(%rdx, %rcx, 3), %r14
6   subq  %r15, %r14
7   shlq  $1, %r14
8   shrq  $1, %r14
9   movq  %rax, (%r15, %r14, 1)
```

Figure 3: Example of shepherded memory access

- **Multi-layer compartmentalisation.** In SGX-Shield, the boundary is fixed. With our critical function protection, we are able to assign multiple boundaries for different security layers to achieve multi-layer compartmentalisation.

While the overall idea of MLIEC is generic and machine-independent, to facilitate a better understanding, we use x86-64 code in this section to provide some concrete examples.

4.3.1 SFI-Based Compartmentalisation

Software Fault Isolation (SFI) aims to prevent code from a lower security layer from accessing the data belonging to a higher security layer. For SFI, we employed SGX-Shield's *shepherded memory access* and *aligned branching*. The method reserves a register (the `R15` register in examples) to store the address of the *boundary* corresponding to the current security layer and instruments code to prevent access to anything below the boundary. Additionally, it prevents bypassing the instrumentation by branching into the middle of the instrumentation code.

Shepherded Memory Access (SMA). The SMA mechanism instruments all memory access in the binary to ensure access occurs above the boundary. In a binary, there are two types of memory access: fixed-address and indirect. Fixed-address accesses have the addresses fixed in the instruction that can't be changed throughout the execution. Indirect memory accesses, however, dereference addresses in registers. Therefore, an indirect memory access can be hijacked by a piece of malicious code while a fixed-address one won't. We focus on the *indirect memory access* and the goal is to force all indirect memory accesses to happen above the boundary register.

Like SGX-Shield, we instrument these indirect memory accesses by converting these addresses into offsets related to the boundary and simply forcing these offsets to be positive with fast bit-wise operations. We reserved `R15` to store the boundary and prohibited its use in code generation. We also reserved `R14` to work on these addresses. We copy the address into `R14` and subtract it with `R15` to get the offset. We then clear the top bit of `R14`, forcing the offset to be positive. By adding the value of `R15`, a benign memory access will obtain the same address as before the operation, while a malicious access will get a different address above the boundary, resulting in reading a random location above the boundary (which, by design, is allowed because it is at

```
// Before:
  jmpq  *%rax

// After:
  andq  %rax, ~32
  jmpq  *%rax
```

(a) Example of indirect branching

```
// Before:
  retq

// After:
  popq  %r14
  andq  %r14, ~32
  jmpq  *%r14
```

(b) Example of return instrumentation

Figure 4: Examples of aligned branching

a lower security level) or simply crashing. An example of this mechanism is illustrated in Figure 3. Note that for a user space x86-64 enclave, there is no overflowing issue with this method since the boundary is in the user space below `0x7FFFFFFFFFFFFFFF` and the offsets are positive.

A special case involves handling `RSP`, the stack pointer register. `RSP` is used to access on-stack local variables with small positive offsets or to be used by certain instructions (e.g., `ret`, and `pop`) to move upwards (except for `push`) by less than 8 bytes. As long as the register itself is above the boundary, stack accesses are safe. The exception is the `push` and `call` instructions that can move the stack pointer downwards and can impose security risks if it moves below the boundary. This is solved by adding a non-accessible page at the bottom so that a small push cannot jump over the page but will instead crash on it. We instrument all other modifications to `RSP` to enforce its value to be greater than the boundary at all times and leave simple stack accesses uninstrumented.

Aligned Branching. One can easily observe that the SMA code must be executed as a whole to achieve the security guarantee. However, indirect branching can breach such a guarantee. Similar to indirect memory accesses, indirect branching refers to instructions that jump to an address stored in a register. Not only can it be exploited to bypass the SMA, when jumping to the middle of a single instruction, the instruction will be misinterpreted. By rewriting the register, an attacker can hijack the control flow and bypass the SMA or perform even more dangerous operations.

To solve this issue, we emit the code into blocks with a fixed-size alignment unit and force an instrumented instruction to be emitted into the same block with the instrumentation code. For every starting point of a basic block (e.g., jump targets, function entry, etc.), we also align them to the alignment unit to ensure the proper functionality of a benign indirect branching. We then instrument every indirect branching to have its target aligned to the alignment unit. This ensures that the SFI instrumentation is enforced and the target is always a correct instruction. With aligned branching, SMA instrumentation cannot be bypassed and no instruction misinterpreting can happen. An example of such instrumentation using a 32-byte unit is illustrated in Figure 4a.

Another typical control flow hijacking attack is Return-Oriented Programming (ROP). This is done by modifying the return pointer stored on stack to cause the code to branch into arbitrary locations. We prevent this by replacing every `ret` instruction with an instrumented indirect branching, as illustrated in Figure 4b. Since every return target will lead a new basic block that is aligned, the return will work as expected.

```
// Basic block 1:
.align 32
  ud2
  ...
  jmp  l1
// Basic block 2:
.align 32
  ud2
l1:
  ...
```

(a) Example of an unaligned critical function

```
.align 32
aligned_wrapper_strlen:
  jmp unaligned_strlen

.align 32
  ud2
unaligned_strlen:
  mov HIGH_BDRY, %r15
  set_high_ctx
  ...
  set_low_ctx
  mov LOW_BDRY, %r15
  ret
```

(b) Example of an aligned wrapper and boundary transition

Figure 5: Examples of unaligned critical function

4.3.2 Unaligned Critical Functions

While our instrumentation is enough to secure most in-enclave functions, there are special critical functions that demand even stronger protection since they either cannot be protected with the instrumentation alone or should not be instrumented due to two primary reasons: *security* and *performance*.

- **Security:** The instrumentation will be insufficient for certain security-critical instructions. For functions containing instructions that grant a page with RWX permissions (e.g., `emodpe` in Intel SGX) or setup/transition the boundary (modifying the `R15` register), a comprehensive security check is needed. However, these checks can be lengthy and may not fit within an alignment unit, making them susceptible to bypassing even when protected using the aligned branching technique.
- **Performance:** For certain regularly used but memory-intensive functions (e.g., `memcpy`), instrumenting them can cause significant performance degradation. While they can be manually modified to have security checks, the checks must be executed together with the function as a whole to guarantee the security. Thus, similar to functions containing security-critical instructions, instrumentation alone is insufficient to protect these functions.

For these functions, we aim for their exectuion as a whole without any hijacked code jumping into the middle. For security critical functions, we even do not want any code to call them indirectly.

Control Flow Integrity (CFI) techniques can be applied to achieve this. Conventionally, CFI is achieved by trapping and checking [37] [38], which imposes significantly performance penalty due to comparing and branching. We took the concept and designed a fast CFI mechanism by applying the aligned branching's property in an opposite direction. Since the code

can only be hijacked to aligned addresses, the key observation here is that if we emit these functions 'unaligned', a piece of malicious code will not be able to hijack the control flow into them. We refer to this technique as the *unaligned critical function* technique.

Rather than trapping and checking, we perform a trapping and faulting by emitting a `ud2` instruction before the code of each block in these functions, which will cause a `#UD` fault and crash the execution. As a result, any instrumented code attempting to perform an indirect branch into these blocks will hit the `ud2` and crash. For a function with multiple basic blocks, we can chain them together using direct `jmp` instructions. An example of two basic blocks with this technique is described in Figure 5a. All of the basic blocks starts with an `ud2` (line `1` and `8`) to prevent the instrumented code from branching into. At the end of the basic block, it chains to the next basic block using a `jmp` (line `5`). By doing this, the only way to call these functions is to do direct calling which is fixed in protected code binary at compile time.

The execution of these critical functions can be interfered by a piece of malicious code if we are still using the original stack. To prevent this, their stack must be swapped to a secured stack under any boundary. For a function that is not meant to be called indirectly, emitting it unaligned is enough. For a function that can be called indirectly but has to be executed as a whole, we can write an aligned wrapper to make a direct call.

For a layer transition, it can be protected using the unaligned critical function technique so that once control flow enters the transition, it is out of the control of regular code. In the transition, the boundary register `R15` is changed to the new boundary and then continues to the high-security function. The stack and heap have to be swapped to the new layer's corresponding ones to prevent an adversary from hijacking. When the high-security function returns to the transition function, the transition function will set the boundary register, stack and heap back before returning to regular code. An example of an aligned wrapper and a layer transition is shown in Figure 5b.

With this technique, we can ensure that these critical functions must be executed as a whole. No code can be hijacked into the middle of them and their security checks cannot be bypassed.

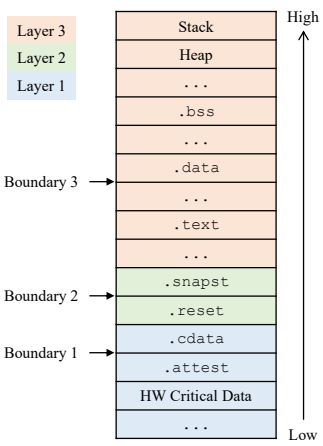
4.3.3 Multi-Layer Compartmentalisation

As discussed above, we need more than two security layers for regular code, reset module, and the nested attestation module. A single-fixed boundary is inadequate to meet this demand. Consequently, we propose a multi-layer compartmentalisation technique to support multiple security layers within a single enclave.

The design of this mechanism begins with using a customised linker script to place segments in layers according to their security level and mark out the boundaries with symbols.

```
1  ...
2  SECTION
3  {
4   ...
5   .attest
6   __boundary_1 = .;
7   .cdata
8   .reset
9   __boundary_2 = .;
10  .snapst
11  ...
12  .text
13  ...
14  __boundary_3 = .;
15  .data
16  ...
17  .bss
18  ...
```

(a) Example of an excerpt from the linker script



(b) The layered layout of an enclave

Figure 6: An enclave layout example with multiple layers

The higher the security level, the lower in the address space it will be placed. The security levels are determined according to specific design, and there is no limitation on the number of layers that can be added. An example of such a linker script is illustrated in Figure 6a, and the layout generated from the linker script is shown in Figure 6b. In this script, there is a `.attest` segment to store the attestation code and `.cdata` to store the critical secret and they are placed in the first layer. The reset module and snapshot (`.reset` and `snapst`) are placed in the second layer. The regular code and data (`.text`, etc.) are placed in the third layer. Each of the `__boundary_x` marks the memory access boundary of a layer. Note that the boundary is marked in the middle of a layer. This is because we limit the access only to the data portion.

Hardware architectural critical data should be placed in the most secure layer to prevent malicious code from modifying. For example, the Save State Area (SSA) storing a thread's context in Intel SGX should be considered as critical.

The security layer of a thread is determined based on which enclave call the thread called and the boundary register `R15` is configured accordingly. We instrument all functions in every layer except for those exempted and protected using the unaligned critical functions mechanism. By doing this, even if lower security level code branches into a higher one, it still cannot violate the boundary since `R15` is not changed.

By combining our snapshot-and-rewinding mechanism with the MLIEC, we are able to provide a fast and secure reset to reuse the enclave while preserving the security guarantees, assuming that the code remains unchanged.

4.3.4 Dynamically Loaded Code

The approach discussed so far works well for statically loaded code, with the assumption that the binary code is unchanged. However, the ability to load code dynamically

can impose significant security concerns over the approach. Clearly, a piece of dynamically loaded code must at least be instrumented to maintain our SFI guarantees. However instrumenting the code alone is not enough.

The concern is that the loader used during dynamic loading must have the ability to grant a page executable permission (e.g., using SGX's `emodpe` instruction), and an adversary can take advantage of this. A simple solution to this problem is to completely disable the feature of dynamic loading and eliminate the permission granting function. However, for serverless scenarios, Ahead-of-Time (AoT) pre-compiled binaries are necessary to achieve a high performance, and we must maintain and secure this feature.

We identified two criteria to enable the dynamic loading that complies with the security requirement in our design:

- **Protected Permission Granting:** The permission granting function must be protected to refuse a re-entrancy request.
- **Equally-Treated Instrumentation:** The loaded code must be instrumented using the same compiler technique.

The first criterion can be achieved by using the unaligned critical function mechanism with proper security checks. The second one requires a mechanism to verify that the binary is instrumented. The exact security check and verification approach has to be tailored to specific needs. We will demonstrate how we achieved these criteria in our prototype in section 5.

5 Implementation

In this section, we present a prototype implementation of the reusable enclave integrated into a full-stack confidential serverless system, and show how we solved the security challenges in a real world scenario. We implemented our system with 10,188 lines of code. The source code will be made available on GitHub.

The implementation starts with an LLVM-based compiler toolchain to implement the MLIEC. We then selected the WebAssembly on OpenWhisk (WOW) [12] project that has already ported a non-confidential version of WAMR to OpenWhisk. We modified the WOW project and integrated our instrumented Intel SGX-enabled WAMR with enclave reset to the system. As a control, we also adapted an uninstrumented SGX-enabled WAMR compiled out of a vanilla LLVM compiler to the system to show the performance overheads of our system. The overall architecture of the system is illustrated in Figure 7.

5.1 The Toolchain

Our compiler toolchain is based on LLVM and has MLIEC implemented as a backend pass. We implemented the system based on LLVM 11.0 with 1,070 lines of C++ code. The implementation provides full support for instrumenting the Intel SGX SDK. To the best of our knowledge, due to complex dependencies, our implementation is the first one to instrument the whole Intel SGX SDK. This allows our MLIEC to be applied to a wide spectrum of existing SGX-SDK-based high-level programming language runtimes.

5.2 The WAMR Enclave

The WAMR enclave consists of a WAMR runtime library with ECalls to expose the functionality of the runtime. We modified 1,457 LoC to the untrusted portion of the WOW's executor to adapt it to an SGX-protected runtime. For the WAMR enclave, the WAMR runtime library was ported to our system with no change except for some alignment markups in the `invokeNative` portion, which uses assembly code. Apart from code of WAMR, we added a total of 4,098 LoC, among which 704 LoC for exposing WAMR APIs, 179 LoC for the enclave reset module and nested attestation module, and 3,215 for SGX's remote attestation.

We compiled the WAMR runtime with MLIEC and integrated the reset module and attestation module into the enclave. The WAMR runtime as well as Intel SGX SDK is placed in the lowest security layer (Layer 1) as shown in Figure 7, while the reset module and the attestation module are placed in Layer 2 and 3, respectively.

To eliminate any useful attack vectors from the untrusted world, we simplified the exposed ECalls of the WAMR runtime into a single one to cover the entire execution procedure and the reset. This approach prevents the untrusted code from rearranging the calling sequence or bypassing the reset. We hard-coded all parameters except for the pointer of the user workload , the error buffer, and their sizes.

5.3 MLIEC in WAMR Enclaves

While we have instrumented the whole enclave and SDK with MLIEC, since WAMR is an execution engine for WASM code, we must carefully consider the potential actions of malicious WASM code. WASM code can be either compiled into bytecode to be interpreted or into x86-64 machine binary for native execution, also known as Ahead of Time Compilation (AoT). The system must not be compromised in either mode.

Interpreter Mode. Protection in interpreter mode is straighforward. Since the interpreter is instrumented, any WASM bytecode interpreted in the WAMR enclave will not be able to compromise security.

AoT Mode. As a native code binary, the AoT binary to be executed may contain any instruction. AoT is essentially a dynamic loading feature, and as discussed in section 4.3.4, we must meet two criteria: *Protected Permission Granting* and *Equally-Treated Instrumentation*.

For the Protected Permission Granting criterion, the key observation is that the dynamic loading feature is for the use

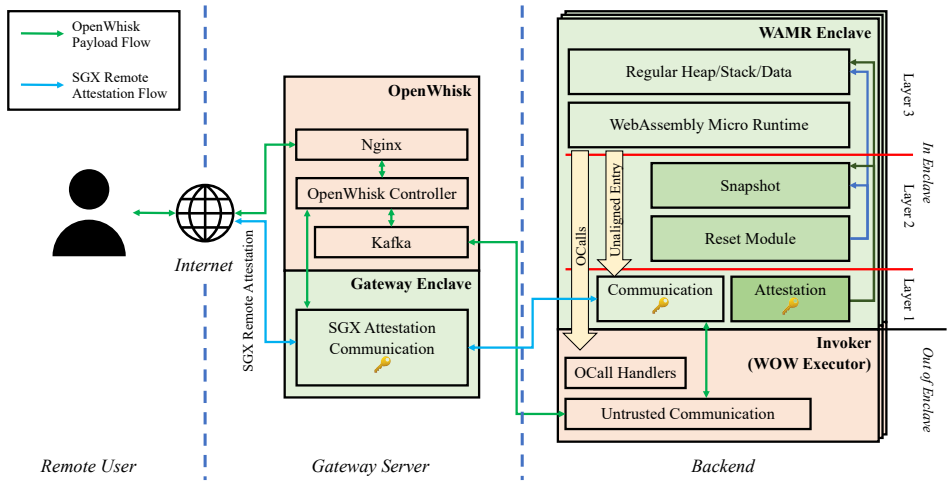


Figure 7: The architecture of the system

of the runtime instead of the user code. After loading the user code, since the runtime no longer use this function, we can disable the `emodpe` before the execution of the user code. To accomplish this, we added a Boolean as a *fuse* in the reset module's security layer (Layer 2), where user code (Layer 3) cannot modify. Once the loading is completed, the fuse Boolean will blow (be set to `false`) before any user code is executed. We then modified the `emodpe` function in SGX SDK to check this fuse and crash when the fuse is blown. We protect the `emodpe` function using the unaligned critical function technique so that the check cannot be bypassed. We use the reset module to set the fuse back after an enclave reset.

The Equally-Treated Instrumentation criterion can lead to two sub-problems: How we instrument the AoT binary and how we verify that a binary is instrumented when we load it. The first problem has a natural solution: We already have an LLVM toolchain to perform the instrumentation, and we can modify the AoT compiler `wamrc` to use our LLVM as the backend to generate instrumented x86-64 binary code. The generated AoT binary will be instrumented using the same MLIEC technique, ensuring that loading it will not break the security of the WAMR enclave.

The second problem is less straightforward. Statically validating a random AoT binary with MLIEC instrumentation would be too costly. However, we can require an AoT binary to be *notarised* to conform to our security standards. Since we already know that an AoT binary compiled using our toolchain meets the security requirement, the serverless platform provider can offer a trusted compilation and signing service where users can upload their WASM bytecode. The service will use our toolchain to properly compile the bytecode into instrumented AoT binaries and sign them. With the signature, the WAMR enclave can verify that the AoT binary is indeed instrumented and can be safely executed. As long as the WAMR enclave only executes AoT binaries signed by the service, its security can be guaranteed. The details of making the service fully protected and attestable have been researched and discussed. Techniques from existing works like [39] can be used.

5.4 End-to-End Trust

The concept of confidential computing naturally distrusts all the communication processes unless verifiable. To ensure the confidentiality of the workload deployment, and establish an end-to-end chain of trust, we designed and implemented a component called *gateway enclave* on the gateway server. The OpenWhisk project was modified with 107 LoC to connect with the gateway enclave. The gateway enclave itself has 1,478 LoC in the untrusted portion and 1,978 LoC in the trusted portion. The gateway enclave is a cryptographic service, providing two functionalities: Delegated Remote Attestation and Workload Relaying.

Delegated Remote Attestation. The gateway enclave will perform attestation to all the worker WAMR enclaves and is then self-attested by a remote user. Using Intel SGX's remote attestation, the gateway enclave can ensure that all the WAMR enclaves associated with it are genuine. When the remote user performs remote attestation to the gateway enclave, they can not only verify that the gateway enclave is genuine and protected, but also trust all the WAMR enclaves associated with the gateway enclave.

Workload Relaying. During the remote attestation process, the gateway enclave exchanges secret keys with the WAMR enclaves, and the user to establish secure communication channels. The user can use the secret key to encrypt the workload and send it to the OpenWhisk service. The OpenWhisk service will request the gateway enclave to re-encrypt the workload using the secret key exchanged with the chosen WAMR worker enclave, and then sends the re-encrypted workload to the worker. With the gateway enclave, a user can

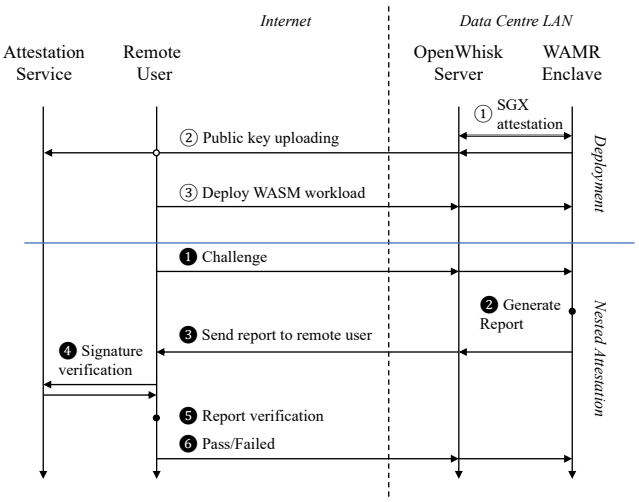


Figure 8: The nested attestation protocol

be assured that the executor enclave is trustworthy and their workloads are always encrypted outside an enclave.

5.5 Nested Attestation for WAMR

With Intel SGX's attestation, the user can verify that the gateway OpenWhisk server is genuine and establish a secure channel with it by exchanging a secret key. The gateway OpenWhisk server will also delegate the user to do the same with the WAMR executor enclaves, ensuring that the actual executors are also trusted, as discussed in section 5.4.

However, in addition to checking if executors are genuine, the reset must also be verifiable to the user. A remote user might also want to verify that the workload is properly deployed and then provision secrets into the workload to establish a direct security channel. We proposed the scheme *nested attestation* and discussed the infrastructure in section 4.2. The private-public key pair generated by the WAMR enclave will be the root of the trust. In this subsection, we discuss a specific protocol for the WAMR enclave.

We illustrate the nested attestation protocol in solid circled numbers in Figure 8 as well as the preparations made during the system deployment in hollow circled numbers. The nested attestation protocol begins with the remote user's challenge to the workload in the WAMR enclave (step ❶). When the workload receives the challenge, it requests the attestation module to generate a report (step ❷). The report contains the hash of the workload, in addition to the contents discussed in section 4.2 for the workload integrity check. A customisable data field is also provided for the workload to include data to be signed, which can be used for key exchange algorithms, etc. The report is then sent to the remote user (step ❸). The remote user submits the report to the attestation service offered by the serverless platform provider to verify the signature (step ❹). If the attestation service confirms that the signature is valid, the remote user will then validate the workload's hash (step ❺) and, if needed, confirm the snapshot's hash from the attestation service. If the check passes, the nested attestation procedure is complete and the enclave is trusted (step ❻).

6 Security Analysis

In this section, we analyse the security of our system. When executing workloads inside the enclave, as long as the enclave environment stays intact, the workloads' security can be guaranteed. The execution security of the confidential serverless has been extensively discussed in [5–7] and therefore we focus our discussion on the security impacts introduced by our design. In our model, an attacker can be from outside (OS, network, etc.), inside (malicious workloads), or both when they collude. We analyse on each step shown in Figure 2 to demonstrate possible attacks and how our system can defeat them.

6.1 Cold Start

The cold start (step ❶) is the simplest case. In this scenario, since there is no involvement of malicious workload, the only possible attacker is from the outside. During step ❶ when we setup the enclave, every operation will be recorded in the initialisation hash and an enclave with a mismatched hash will be refused to be launched. The enclave will also be untrusted if it does not pass the remote attestation and snapshoting process. An attacker might also try to execute a malicious workload before taking the snapshot, trying to leave unwanted data in the snapshot. However, the hash of a genuine snapshot that is correctly taken is known, and an incorrect snapshot will result in a hash mismatch during the nested attestation.

6.2 Workload Execution and Result Returning

As described in section 5.2, we only have a single all-in-one execution ECall that handles the whole execution and the return of the results. Therefore, ❷ and ❸ are discussed together.

Outside Attacker: Reentering ECalls. There are only two ECalls exposed other than ECalls of the SGX's remote attestation: the snapshot ECall and the all-in-one execution ECall that performs the execution and reset all together. For the snapshot ECall, reentering it other than the initial state will cause the snapshot to have a different hash, which can be detected in the nested attestation procedure because the hash in the signed report contains is different from a genuine snapshot hash, causing the remote user to reject the execution.

For the execution ECall, it only takes the two untrusted buffers and their sizes, leaving no procedure rearranging or faulty parameter attacks. Reentering it will cause a fault since an execution is already happening.

Inside Attacker: Steal or modify security-sensitive data. A malicious workload may try to utilise runtime bugs to

compromise the environment. However, as the runtime is instrumented and the workload is either a bytecode interpreted by the runtime or an instrumented AoT binary, it will not be able to read or write below its layer boundary. This means that the snapshot and reset counter in the reset module and the signing key in the attestation module may never be read/write by a workload.

Inside Attacker: Gadget deployment. To deploy a gadget, a malicious workload must be able to edit existing executable pages or grant writable pages with executable permission by reentering `emodpe`. Since the `emodpe` instruction is protected with the fuse Boolean as described in section 5, and all code pages are protected as read-only, a malicious workload can neither modify the WAMR runtime nor itself. It also cannot inject any gadget since it is not possible to grant a page with executable permission without the `emodpe`.

Collusion: Architectural. An outside attacker may want to collude with a malicious workload to attack the system using architectural behaviours. The way to do this is to trigger an interrupt to the enclave so that the states are saved to the memory and then let the malicious workload modify the states. For example, an attacker may want to modify the boundary register `R15` in the saved state, essentially allowing it to lower the boundary to 0 to read and write any data. However, since SSA and architectural security-sensitive data are protected in the most secure layer, they may never be modified by a malicious workload.

6.3 Environment Reset

The reset procedure ❹ is integrated into the all-in-one execution ECall and is performed automatically before the next execution as described in section 5.2.

Outside Attacker: Delayed resuming. When a workload is multi-threaded, an outside attacker may want to issue an interrupt to an executor enclave thread during the execution of a workload and delay the resuming after main thread performs a reset to cause unexpected effects. However, this is not possible on our system since the reset will not be performed unless the workload is totally finished.

Inside Attacker: Persisting changes. A malicious workload may have changed the runtime's configuration and want to persist the changes. However, as our reset mechanism overwrites all data fields within the data segments and zeros out the heap and stack, it is not possible for the malicious workload to leave any changes within the runtime memory.

7 Evaluation

In this section, we present our evaluation results to demonstrate the performance overhead imposed and to show our performance benefits for serverless execution scenarios. We tested our system from multiple aspects to show the performance improvement over relaunching the entire enclave, as well as the overhead impact of our instrumentation. We also compared our system with existing solutions using workloads that can reflect real-world serverless characteristics. The platform we used was an Intel Xeon Gold 5318Y CPU running at 2.1 GHz with 32 GiB of RAM. The OS is Ubuntu 20.04 LTS with a stock Linux 5.11 kernel. AoT binary blobs are compiled and instrumented with the same LLVM for our implementation and with a vanilla Clang sandboxed using `-boundary-check` as control. We present the result of our experiments with an alignment unit of 32 bytes. We will also discuss the performance implications of different alignment unit sizes in this section.

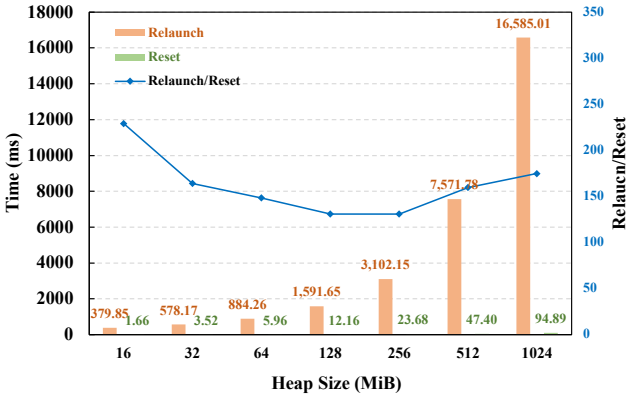


Figure 9: Time spent in relaunching and resetting. The blue line shows the ratio of relaunching time over resetting time.

7.1 Launching vs. Resetting

The relaunching time and resetting time of a WAMR enclave varies depending on the enclave size, particularly the heap size. We illustrate the relaunching time of a vanilla WAMR enclave and the resetting time of our instrumented WAMR enclave in Figure 9. On average, resetting is 159× faster than launching an enclave. This is because the instructions `eadd` and `einit` used during the launching process are very time consuming. For resetting, in contrast, rewinding the data segments takes only 87 μs on average, and the main overhead comes from zeroing the heap. On modern x86-64 architecture, performing `memset` to zero the heap using AVX2 or even simply `rep stos` can be very fast, saving ranged from 378.19 ms on the 16 MiB side all the way up to 16.5 seconds compared to relaunching.

In addition to the launching time, an SGX enclave must be attested using the remote attestation each time after launching to be trusted. We chose the fastest DCAP method for the evaluation and the average overhead of it was 25 *ms*. However, these overheads are too small compared to the relaunching time and will not make a significant impact on the launching.

7.2 Instrumentation Overhead

To evaluate the instrumentation overhead, we chose NBench [40] as our subject. We tested NBench in multiple execution modes of the WAMR runtime: the AoT mode, the classic interpreter mode, and the fast interpreter mode. In Figure 10, we show the ratio of the NBench score acquired by a vanilla WAMR runtime in SGX and ours that was instrumented with MLIEC. Therefore, the figure shows how many times faster the vanilla version runs than our version, reflecting the slowdown of the MLIEC instrumentation.

Instrumentation Overheads of AoT Mode. In AoT mode, we compiled the workload with our LLVM instrumentation on our system while using WAMR's stock boundary check sandboxing in the vanilla version. The geometric mean of the overhead is 20%. One can see that most of these benchmarks have an overhead of 10% or below compared to the vanilla version. Our instrumentation even made ASSIGNMENT, NEURAL NET, and LU DECOMPOSITION perform faster than WAMR's AoT sandboxing.

However, the FOURIER benchmark was showing significant performance degradation with our instrumentation. Our investigation revealed that the FOURIER benchmark highly depends on the C math library, resulting in excessive native C calls during the execution. The native call wrapper of WAMR is memory intensive due to ABI translation mechanism from WASM to native x86. The mechanism encodes C function prototypes as strings and parses the string when a native C function is called. Parsing these strings and arranging the arguments in memory requires massive variable pointer dereferences which are instrumented using SMA. The geometric mean of the rest of the benchmarks' (without FOURIER) overheads is only 5%.

Instrumentation Overhead of Interpreter Modes. In WAMR, there are two interpreter modes: the classic mode and the fast mode. Note that in the interpreter mode we run the same WASM bytecode for both instrumented and vanilla, since the security is guaranteed by the interpreter.

In the classic mode, the interpreter implements a standard stack machine. The overhead of it (14%) is quite low. In the fast mode, WAMR recompiles the code into a register-based IR bytecode with other optimisations. This resulted in larger memory footprints [41] and a higher average overhead of 32% (see Figure 10). In our system, the instrumentation to memory implies more overheads to workloads that have more pointer dereferences. In WAMR, since the code is stored in a heap and all heap pointer dereferences will trigger SMA, increasing the code size will inevitably increase the overhead.

Alignment Unit Sizes and Performance. To provide insight into the impact of different alignment unit sizes, we also ran NBench with a system sanitised with a 64-byte alignment unit. The result remained unchanged with the same overhead geometric means: 20% for AoT mode, 14% for classic interpreter mode and 32% for fast interpreter mode.

This result is within our expectation, since the main overhead comes from our SFI and DEP instrumentation. Theoretically there is a trade-off between a small and a large alignment unit because it is very hard to exactly fill up an alignment block and there can be a few `nops` at the end of a block. This means that using a larger alignment unit that can have less `nops`. Although this may bring about some fluctuation in the performance of certain workloads at a very microscopic level, our result shows that generally there will be no significant impact.

7.3 Real-World End-to-End Performance

To demonstrate the end-to-end real-world benefits of our system, we evaluated the end-to-end performance using the workloads used in the WOW project [12] (excluding sleeping and non-supported features like file system) and compared our work to a vanilla WAMR executor without reset, as well as existing confidential serverless projects: S-FaaS [5] and AccTEE [7]. The four workloads cover different perspectives used in serverless scenarios, ranged from heavy computation and hashing to light system calls and parameter handling. Note that S-FaaS used Duktape embedded JS engine which is not capable of performing complicated tasks like hashing and finding large prime numbers. The executors of the two existing projects are set to be relaunched after each execution.

The enclave was set to have a heap size of 256 MiB, in the middle between the industry's 50 percentile (170 MiB) and 90 percentile (400 MiB) in terms of serverless workload memory consumption according to [10] [11]. For WAMR executors, we broke down the execution time into Reset, Load, Instantiate, Create Environment, Function Lookup, Execution and the Communication overheads as well as the End-to-End Encryption and Report Generation overheads exclusive to our system. For S-FaaS and AccTEE, due to the architectural difference, we broke down the time into Reset, Init, and Execution. The reuslt is shown in Table 1 and illustrated in Figure 11.

The result showed that our system can significantly improve end-to-end performance compared to others. For the vanilla version and the two existing solutions, the overhead of relaunching the enclave claimed most of the end-to-end time, resulting in massive delays and wasting a huge amount of computation resources. In our system, the communication costs dominated over the reset, however, in a much less significant way compared to relaunching. Our end-to-end encryption and nested attestation's report generation only took a fraction of the overall overhead.

7.4 Summary of Results

Reset Benefit. Recent studies [10] [11] have shown that in real-world applications, more than 50% of serverless

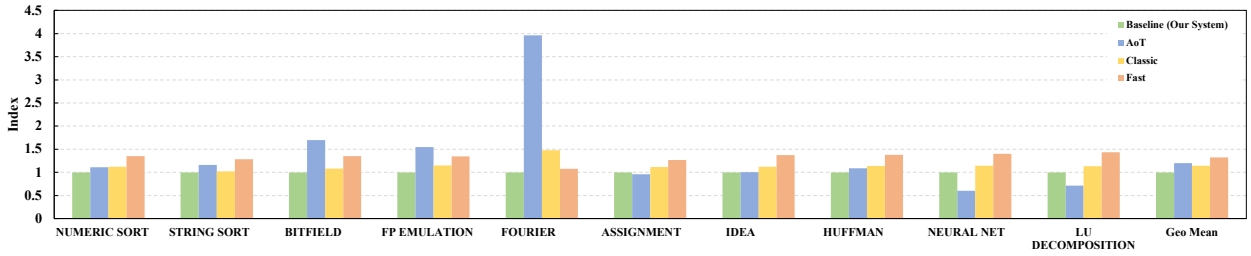


Figure 10: The normalised NBench result

		Reset	Load/Init	Instantiate	Create Env	Lookup	Execution	Communication	Encryption	Report Gen	Total
Ours	ADD	24.80	3.09	0.27	0.02	0.01	0.08	94.90	9.31	0.82	133.30
	CLOCK	24.21	2.04	0.29	0.04	0.02	0.11	89.59	10.05	0.53	126.88
	HASH	23.79	3.15	0.28	0.04	0.02	0.09	92.63	9.15	0.51	129.66
	PRIME	24.87	2.27	0.33	0.04	0.01	1031.21	85.01	11.24	0.51	1155.49
Vanilla	ADD	3266.52	0.95	0.16	0.01	0.01	0.06	83.80	N/A	N/A	3351.52
	CLOCK	3237.99	0.82	0.15	0.01	0.01	0.08	77.88	N/A	N/A	3316.94
	HASH	3233.70	0.94	0.15	0.01	0.01	0.07	78.51	N/A	N/A	3313.38
	PRIME	3294.87	1.01	0.15	0.01	0.01	818.07	78.91	N/A	N/A	4193.02
S-FaaS	ADD	2246.01	1.02	N/A	N/A	N/A	0.37	N/A	N/A	N/A	2791.63
	CLOCK	2790.22	1.04	N/A	N/A	N/A	0.40	N/A	N/A	N/A	2247.46
	HASH	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	Can't run
	PRIME	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	Can't run
AccTEE	ADD	4589.42	340.20	N/A	N/A	N/A	0.79	N/A	N/A	N/A	4930.42
	CLOCK	4591.99	373.60	N/A	N/A	N/A	0.83	N/A	N/A	N/A	4966.42
	HASH	4597.17	407.86	N/A	N/A	N/A	1.31	N/A	N/A	N/A	5006.35
	PRIME	4596.01	164.48	N/A	N/A	N/A	2075.07	N/A	N/A	N/A	6835.58

Table 1: The end-to-end execution time breakdown (in milliseconds)

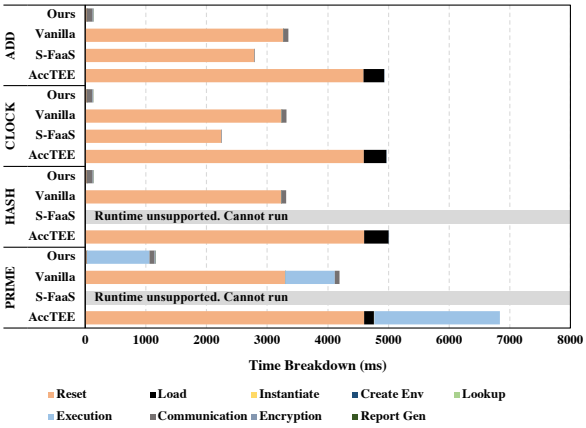


Figure 11: The end-to-end execution time breakdown

workloads finish under 1 second on average. These studies also showed that 50% of serverless workloads take less than 170 MiB of memory and 90% take less than 400 MiB of memory. Looking at Figure 9, we can easily observe that the overhead of relaunching on its own is more than 2 times the execution time even for a smaller 170 MiB enclave.

Note that while the memory can be dynamically expanded in the latest SGX2, the expansion is even slower than having the memory at launching [24], resulting in an even higher overhead. This means that for the best performance, the enclave should have a heap that can accommodate the maximum workload it can handle, no matter how much memory a *real* workload running on it takes. However, larger enclaves will impose even higher launching overheads on small and fast serverless workloads compared to our resetting mechanism.

Interpreter Mode vs. AoT Mode. To provide the best performance, the AoT mode is clearly the first choice. When comparing the results of NBench, the fast interpreter is 14× slower than AoT while the classic interpreter is 49× slower than AoT. The interpreter modes, even the vanilla fast version, are too slow for real world applications. While the overhead we imposed is acceptable, we do not believe that using the interpreter mode in a production environment is practical. However, the downside of adopting the AoT mode is that it requires additional software tools to enable trusts on the compilers, as described in section 5.3.

Possible Optimisations. Our system can be further improved. For the AoT mode, one of the reasons for the huge overhead introduced in the FOURIER benchmark is native C API calling. We analysed that it is due to the C prototype string parsing causing massive amount of pointer dereferences and triggering many instrumented code paths. To overcome this issue, optimisations can be made to the WAMR's native call mechanism to reduce the pointer dereferences using techniques like caching. For the interpreter mode, the slowdown mainly comes from the instrumentation during the fetching of the bytecode. A viable solution can be achieved by prefetching a large chunk of code onto the stack and accessing them on the stack. Since stack accesses are not instrumented, one can significantly reduce the overhead.

Name	Special Hardware	Compartments		Special SDK?	Commercially Available?	Overhead
		Data	Code			
LIGHTENCLAVE	Intel MPK HC	Multi	Multi	N	N	≈0%?
MPTEE	Intel MPX	Multi	1	N	N	20%
Occlum	Intel MPX	Multi	Multi	Y	N	36%
Spons & Shields	Intel MPX	Multi	Multi	Y	N	22%
Nested Enclave	HC	Multi	Multi	Y	N	2%?
SGX-Shield	None	2	2	Y	Y	14%
CHANCEL	None	Multi	1	Y	Y	12%
Ours	None	Multi	Multi	N	Y	20%

Table 2: Comparision between IEC techniques. '?' means the result is simulated. HC stands for 'Hardware Changes'.

However, these require massive architectural changes to the WAMR and we leave them for future work.

8 Related Work

Confidential Serverless Computing. Closest to our work is the line of research integrating TEEs with serverless computing [5–7]. In particular, S-FaaS [5] also integrated Intel SGX with OpenWhisk to build a confidential FaaS solution. In the industry, commercially available platforms like Conclave [8] are already providing confidential serverless computing service to the public. However, all of the existing solutions have the trade-off of security for practical performance by assuming a perfect implementation of sandboxes or have to relaunch for true security [14].

Confidential Runtime. There are many existing work on protecting a high-level programming language runtime using TEEs. For example, the support of WebAssembly runtime within SGX has been an effort made by many to establish portable and secure two-way sandboxes [7, 42, 43]. Open source projects also include Confidential Computing Consortium's Enarx [44] and Bytecode Alliance's WebAssembly Micro Runtime [34], which is also used in our work.

To build a secure runtime that can be reused, MesaPy [45] tried to use formal verification to build a Python runtime with a high-security sandboxing. Unfortunately, MesaPy is not finished and halts with the last update in 2018.

Intra-Enclave Compartmentalisation (IEC). Partitioning an enclave into two or more compartments can be done using hardware-assisted or software-only techniques. We list prior works on IEC and provide a comparison with them in Table 2.

The first five IEC techniques require hardware support. LIGHTENCLAVES [46] proposed new hardware extensions to support the use of Intel MPK [47], which relies on an untrusted OS to perform permission configurations, to partition SGX enclaves. However, this requires new hardware extensions that are not available on commodity CPUs. Other works have proposed to use Intel MPX [48] to provide IEC, such as MPTEE [49], Occlum [50] and Spons & Shields [51]. However, since MPX is buggy and has been deprecated [48], these solutions no longer work in newer generations CPUs. Moreover, the performance overhead of these solutions (20% for MPTEE, 36% for Occlum, 22% for Spons & Shields) is comparable to ours (about 20%), making the need of special hardware unwarranted. The nested enclave [52] is a new architectural design that enables an inner enclave to run inside an outer enclave, such that the inner enclave may possess a higher security level than the outer enclave. However, it also requires special hardware extensions.

Software-only solutions such as SGX-Shield [27] and CHANCEL [28] can also be used to partition an enclave. SGX-Shield partitioned the enclave into two security levels, where the higher one can access the other but not vice versa. CHANCEL achieved time-sharing a single enclave by multiple users.

Our work is also software-only, but we extended and improved the established methods, particularly SGX-Shield, in the following aspects: ***Multi-Layer Support*** to support multiple layers for hosting code and data with different security levels, ***Unaligned Critical Function*** to protect the control flow of security- or performance-critical functions from being hijacked, and a modern and LLVM-based ***Generic Tool Chain*** that allows existing executor runtimes to be ported directly by instrumenting the SDK together on commercially-available hardware.

Software Fault Isolation (SFI) and Control Flow Integrity (CFI). SFI is designed to contain faulty or even malicious behaviours of the software within certain domain [26]. SFI can be done at multiple levels, commonly at the machine code level [27], [30], [53], [29], [54] or at intermediate high-level representations (IR) [55], [56], [57], [58], [59].

Our work falls into the category of SFI at machine code level. To enumerate some of these studies, NaCl [30] utilised obsolete x86 segment registers to partition an address space into several isolated sandboxes. XFI [29] enables dynamically-loaded code to be verified using a disassembler and verifier, but it lacks support for advanced ISA extensions (e.g., SSE, XMM). XFI uses a compare-and-branching mechanism and is hence slower. Sehr et al. [54] proposed a fast boundary plus offset mechanism that is similar to ours. However, it requires that each compartment be 4 GiB, which is not practical in enclave settings.

For dynamically loaded AoT binaries, SFI verification approaches theoretically can also be used to verify if a binary is properly instrumented. Existing SFI verification approaches usually require a large disassembler code base and are limited in their capabilities.For example, XFI has limitations on modern instruction sets [29]. VeriWASM [60] also implemented a verifier for the SFI generated by the Lucet compiler. However, the Lucet compiler's SFI is done in a coarse-grain way by simply keeping the memory within a 4 GiB range compared to our MLIEC [61]. To the best of our knowledge, building a system that includes a verifier while still maintaining a small TCB as our system would not be practical given that a fully-fledged disassembler must be employed.

Our unaligned critical function technique, while facilitating the SFI mechanism, can also fell into the broad category of

Control Flow Integrity (CFI). Designed for our specific needs, the technique implements a fast and effective trap-and-fault mechanism compared to the traditional trap-and-check mechanism [37].

9 Conclusion

In this paper, we propose reusable enclaves, an innovative technique that enables secure, rapid, and verifiable reuse of SGX enclaves, and apply it to confidential serverless computing in order to mitigate the cold boot latency, which is critical to the performance of serverless functions. To demonstrate its practicality and efficiency, we constructed a prototype serverless cloud system using OpenWhisk, integrating a WebAssembly runtime with reusable enclaves. Our evaluation reveals that this approach significantly reduces cold start latency while imposing a reasonable impact on the performance of standard execution.

Acknowledgment

We thank our shepherd as well as the anonymous reviewers for their insightful comments, which have significantly improved the paper. Shixuan Zhao, Mengya Zhang, and Zhiqiang Lin were partially supported by NSF awards 2112471 and 2207202. Pinshen Xu and Yinqian Zhang were partially supported by Ant Group. Guoxing Chen was partially supported by the NSFC under Grant No. 62102254 and Shanghai Pujiang Program under Grant No. 21PJ1404900.

References

[1] "What is confidential computing?" https://docs.microsoft.com/en-us/azure/confidential-computing/overview, (Accessed on 30/04/2022).

[2] "Confidential computing," https://cloud.google.com/confidential-computing, (Accessed on 30/04/2022).

[3] "TEE-based confidential computing," https://partners-intl.aliyun.com/help/en/container-service-for-kubernetes/latest/tee-based-confidential-computing-tee-based-confidential-computing, (Accessed on 30/04/2022).

[4] "FaaS (Function-as-a-Service)," https://www.ibm.com/cloud/learn/faas, (Accessed on 30/04/2022).

[5] 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, CCS 2019. London, United Kingdom: Association for Computing Machinery, 2019.

[6] W. Qiang, Z. Dong, and H. Jin, "Se-Lambda: Securing Privacy-Sensitive Serverless Applications Using SGX Enclave," in Security and Privacy in Communication Networks. Singapore: Springer International Publishing, 2018.

[7] D. Goltzsche, M. Nieke, T. Knauth, and R. Kapitza, "AccTEE: A WebAssembly-Based Two-Way Sandbox for Trusted Resource Accounting," in Proceedings of the 20th International Middleware Conference, Middleware '19. Davis, CA, USA: Association for Computing Machinery, 2019.

[8] "Build & host privacy-preserving apps with ease," https://www.conclave.net, (Accessed on 07/09/2022).

[9] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing Serverless Platforms with ServerlessBench," in Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC 2020. Virtual Event: Association for Computing Machinery, 2020.

[10] 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. Virtual Event: USENIX Association, 2020.

[11] S. Eismann, J. Scheuner, E. V. Eyk, M. Schwinger, J. Grohmann, N. R. Herbst, C. L. Abad, and A. Iosup, "A Review of Serverless Use Cases and their Characteristics," ArXiv, vol. abs/2008.11110, 2020.

[12] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, "Pushing Serverless to the Edge with WebAssembly Runtimes," in 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing, CCGrid 2022. Taormina, Italy: IEEE, 2022.

[13] "CVE-2023-26489," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-26489, 2023.

[14] "Third SGX Community Day - Intel Communities," https://community.intel.com/t5/Blogs/Tech-Innovation/Data-Center/Third-SGX-Community-Day/post/1393177, (Accessed on 04/02/2023).

[15] "Apache OpenWhisk is a serverless, open source cloud platform," https://openwhisk.apache.org, (Accessed on 11/09/2022).

[16] "AWS Lambda," https://aws.amazon.com/lambda/, (Accessed on 30/04/2022).

[17] "Azure Functions," https://azure.microsoft.com/en-us/services/functions, (Accessed on 30/04/2022).

[18] "Cloud Functions," https://cloud.google.com/functions, (Accessed on 30/04/2022).

[19] "IBM Cloud Functions," https://cloud.ibm.com/functions/, (Accessed on 30/04/2022).

[20] "WebAssembly," https://webassembly.org, (Accessed on 30/04/2022).

[21] "Emscripten," https://emscripten.org, (Accessed on 30/04/2022).

[22] "binaryen | Compiler infrastructure and toolchain library for WebAssembly," http://webassembly.github.io/binaryen/, (Accessed on 30/04/2022).

[23] "WHITE PAPER - Intel® Software Guard Extensions (Intel® SGX)," https://www.intel.com/content/dam/develop/external/us/en/documents/overview-signing-whitelisting-intel-sgx-enclaves.pdf, (Accessed on 30/04/2022).

[24] M. Li, Y. Xia, and H. Chen, "Confidential Serverless Made Efficient with Plug-In Enclaves," in 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture, ISCA 2021. Virtual Event: IEEE, 2021.

[25] O. Weisse, V. Bertacco, and T. 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: Association for Computing Machinery, 2017.

[26] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient Software-Based Fault Isolation," in Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles, SOSP 1993. Asheville, NC, USA: Association for Computing Machinery, 1993.

[27] J. Seo, B. Lee, S.-M. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs," in 24th Annual Network and Distributed System Security Symposium, NDSS 2017. San Diego, CA, USA: The Internet Society, 2017.

[28] A. Ahmad, J. Kim, J. Seo, I. Shin, P. Fonseca, and B. Lee, "CHANCEL: Efficient Multi-client Isolation Under Adversarial Programs," in 28th Annual Network and Distributed System Security Symposium, NDSS 2021. San Diego, CA, USA: The Internet Society, 2021.

[29] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software Guards for System Address Spaces," in 7th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2006. Seattle, WA: USENIX Association, 2006.

[30] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: A Sandbox for Portable, Untrusted x86 Native Code," in 2009 30th IEEE Symposium on Security and Privacy, SP 2009. IEEE, 2009.

[31] "When Does Cold Start Happen on AWS Lambda?" https://mikhail.io/serverless/coldstarts/aws/intervals/, (Accessed on 11/09/2022).

[32] F.-X. Standaert, Introduction to Side-Channel Attacks. Boston, MA: Springer US, 2010.

[33] 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 40th IEEE Symposium on Security and Privacy, SP 2019. IEEE, 2019.

[34] "WebAssembly Micro Runtime," https://github.com/bytecodealliance/wasm-micro-runtime, (Accessed on 10/08/2021).

[35] "Attestation Architecture | Trusted Computing Group," https://trustedcomputinggroup.org/resource/dice-attestation-architecture/, (Accessed on 18/09/2022).

[36] J. Saltzer and M. Schroeder, "The protection of information in computer systems," Proceedings of the IEEE, vol. 63, no. 9, 1975.

[37] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity Principles, Implementations, and Applications," ACM Transactions on Information and System Security, vol. 13, no. 1, nov 2009.

[38] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary Stirring: Self-Randomizing Instruction Addresses of Legacy X86 Binary Code," in Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS 2012. Raleigh, NC, USA: Association for Computing Machinery, 2012.

[39] F. Sang, M.-W. Shih, S. Lee, X. Zhang, M. Steiner, M. Vij, and T. Kim, "PRIDWEN: Universally Hardening SGX Programs via Load-Time Synthesis," in 2022 USENIX Annual Technical Conference, ATC 2022. Carlsbad, CA, USA: USENIX Association, 2022.

[40] "Linux/Unix nbench." https://www.math.utah.edu/~mayer/linux/bmark.html, (Accessed on 29/04/2022).

[41] J. Xu, L. He, X. Wang, W. Huang, and N. Wang, "A fast WebAssembly Interpreter design in WASM-Micro-Runtime." Intel, 2021.

[42] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "Twine: An Embedded Trusted Runtime for WebAssembly," in 2021 IEEE 37th International Conference on Data Engineering, ICDE 2021. Chania, Greece: IEEE, 2021.

[43] M. Nieke, L. Almstedt, and R. Kapitza, "Edgedancer: Secure Mobile WebAssembly Services on the Edge," in Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking, EdgeSys 2021. Virtual Event: Association for Computing Machinery, 2021.

[44] Confidential Computing Consortium, "Enarx: WebAssembly + Confidential Computing," https://enarx.dev/, 2021.

[45] "mesalock-linux/mesapy: A Fast and Safe Python based on PyPy," https://github.com/mesalock-linux/mesapy, (Accessed on 24/09/2022).

[46] J. Gu, B. Zhu, M. Li, W. Li, Y. Xia, and H. Chen, "A Hardware-Software Co-design for Efficient Intra-Enclave Isolation," in 31st USENIX Security Symposium, USENIX Security 2022. Boston, MA: USENIX Association, 2022.

[47] "Memory Protection Keys," https://www.kernel.org/doc/html/latest/core-api/protection-keys.html, (Accessed on 10/10/2022).

[48] "Intel Memory Protection Extensions Enabling Guide," https://www.intel.com/content/www/us/en/developer/articles/guide/intel-memory-protection-extensions-enabling-guide.html, (Accessed on 10/10/2022).

[49] W. Zhao, K. Lu, Y. Qi, and S. Qi, "MPTEE: Bringing Flexible and Efficient Memory Protection to Intel SGX," in Proceedings of the 15th European Conference on Computer Systems, EuroSys 2020. Heraklion, Greece: Association for Computing Machinery, 2020.

[50] 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 Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2020. Lausanne, Switzerland: Association for Computing Machinery, 2020.

[51] V. A. Sartakov, D. O'Keeffe, D. Eyers, L. Vilanova, and P. Pietzuch, "Spons & Shields: Practical Isolation for Trusted Execution," in Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2021. Virtual Event: Association for Computing Machinery, 2021.

[52] J. Park, N. Kang, T. Kim, Y. Kwon, and J. Huh, "Nested Enclave: Supporting Fine-grained Hierarchical Isolation with SGX," in 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture, ISCA 2020. Salvador-BA, Brazil: IEEE, 2020.

[53] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC Architecture," in 15th USENIX Security Symposium, USENIX Security 06. Vancouver, B.C. Canada: USENIX Association, 2006.

[54] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting Software Fault Isolation to Contemporary CPU Architectures," in Proceedings of the 19th USENIX Conference on Security, USENIX Security 2010. Washington DC, USA: USENIX Association, 2010.

[55] A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe, "Efficient and Language-Independent Mobile Programs," in Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation, PLDI 1996. Philadelphia, PA, USA: Association for Computing Machinery, 1996.

[56] G. Morrisett, K. Crary, N. Glew, and D. Walker, "Stack-Based Typed Assembly Language," Journal of Functional Programming, vol. 12, no. 1, jan 2002.

[57] V. Prasad, W. Cohen, F. Eigler, M. Hunt, J. Keniston, and J. Chen, "Locating system problems using dynamic instrumentation," in 2005 Ottawa Linux Symposium, OLS 2005, Ottawa, ON, Canada, 2005.

[58] B. Ford and R. Cox, "Vx32: Lightweight User-level Sandboxing on the x86," in 2008 USENIX Annual Technical Conference, ATC 2008. Boston, MA: USENIX Association, 2008.

[59] "The Java Virtual Machine Specification," https://docs.oracle.com/javase/specs/jvms/se19/html/index.html, (Accessed on 10/10/2022).

[60] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Доверяй, но проверяй: SFI Safety for Native-Compiled WASM," in 28th Annual Network and Distributed System Security Symposium, NDSS 2021. San Diego, CA, USA: The Internet Society, 2021.

[61] "bytecodealliance/lucet: Lucet, the Sandboxing WebAssembly Compiler." https://github.com/bytecodealliance/lucet, (Accessed on 11/10/2022).