Hardware Hardened Sandbox Enclaves for Trusted Serverless Computing

JOONGUN PARK, KAIST, Republic of Korea
SEUNGHYO KANG, KAIST, Republic of Korea
SANGHYEON LEE, KAIST, Republic of Korea
TAEHOON KIM, ETRI, Republic of Korea
JONGSE PARK, KAIST, Republic of Korea
YOUNGJIN KWON, KAIST, Republic of Korea
JAEHYUK HUH, KAIST, Republic of Korea

In cloud-based serverless computing, an application consists of multiple functions provided by mutually distrusting parties. For secure serverless computing, the hardware-based trusted execution environment (TEE) can provide strong isolation among functions. However, not only protecting each function from the host OS and other functions, but also protecting the host system from the functions, is critical for the security of the cloud servers. Such an emerging trusted serverless computing poses new challenges: each TEE must be isolated from the host system bi-directionally, and the system calls from it must be validated. In addition, the resource utilization of each TEE must be accountable in a mutually trusted way. However, the current TEE model cannot efficiently represent such trusted serverless applications. To overcome the lack of such hardware support, this paper proposes an extended TEE model called *Cloister*, designed for trusted serverless computing. *Cloister* proposes four new key techniques. First, it extends the hardware-based memory isolation in SGX to confine a deployed function only within its TEE (enclave). Second, it proposes a trusted monitor enclave that filters and validates system calls from enclaves. Third, it provides a trusted resource accounting mechanism for enclaves which is agreeable to both service developers and cloud providers. Finally, Cloister accelerates enclave loading by redesigning its memory verification for fast function deployment. Using an emulated Intel SGX platform with the proposed extensions, this paper shows that trusted serverless applications can be effectively supported with small changes in the SGX hardware.

CCS Concepts: • **Security and privacy** → **Hardware-based security protocols**.

Additional Key Words and Phrases: Security, Hardware, Serverless computing, Trusted Execution Environment

1 INTRODUCTION

Serverless computing has become a mainstream cloud service with the advent of platforms such as AWS Lambda, Azure Function, and Google Function [25, 48, 66]. In the serverless computing, the cloud provider manages server

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) (IITP2017-0-00466 SW StarLab and IITP2021-0-01817 Development of Next-Generation Computing Techniques for Hyper-Composable Datacenters). This work was also supported by the National Research Foundation of Korea (NRF-2022R1A2B5B01002133). Both grants are funded by the Ministry of Science and ICT, Korea.

Authors' addresses: Joongun Park, jupark@casys.kaist.ac.kr, KAIST, Daejeon, Republic of Korea, 34141; Seunghyo Kang, shkang@casys.kaist.ac.kr, KAIST, Daejeon, Republic of Korea, 34141; Sanghyeon Lee, jupark@casys.kaist.ac.kr, KAIST, Daejeon, Republic of Korea, 34141; Taehoon Kim, taehoon.kim@etri.re.kr, ETRI, Daejeon, Republic of Korea, 34141; Jongse Park, jspark@casys.kaist.ac.kr, KAIST, Daejeon, Republic of Korea, 34141; Youngjin Kwon, yjkwon@casys.kaist.ac.kr, KAIST, Daejeon, Republic of Korea, 34141; Jaehyuk Huh, jhhuh@casys.kaist.ac.kr, KAIST, Daejeon, Republic of Korea, 34141.

1544-3566/2023/11-ART
https://doi.org/10.1145/3632954

infrastructures, and developers compose services with functions running in containers. The serverless model allows cloud providers to optimize the platform using their customized software stacks while developers can focus on their services without concerning deployment, scalability, and failure resistance of services.

In serverless computing, a service is composed by integrating fine-grained functions written by different developers, and each function communicates with each other according to APIs provided by the platform. Compared to traditional cloud architectures, the serverless model has a more complex relationship among mutually distrusting parties: clients who need to protect their data, service developers who require the isolation of each function, and the platform provider who wants to protect their system from malicious clients and developers. To meet the security requirements, a new protection model has been proposed for the complex scenarios, called *trusted serverless computing model* [23, 47, 51, 85]. In the trusted serverless computing, each function is encapsulated by a protection domain to isolate the function from the platform.

Prior approaches [22, 47, 55, 76] use hardware-based Trusted Execution Environment (TEE) which enables the strong isolation of functions in remote clouds, even when the servers are exposed to potential vulnerability in privileged software and physical attacks. Recent TEE supports such as Intel Software Guard Extension (SGX) and RISC-V Keystone [61] provide fine-grained isolated execution environments called *enclaves* protected by the CPU hardware. The CPU hardware isolates each enclave (function) from the operating system (the platform) and other applications (other functions). Its code and data can be encrypted and integrity-verified while they reside in the external DRAM.

However, trusted serverless computing requires more protections and services which cannot be supported by the traditional TEE: First, in the traditional TEE, the codes inside an enclave can freely access or jump to the remaining untrusted memory of the process. Such uni-directional protection can endanger the rest of the system if the enclave code is malicious [89]. To address such vulnerability, the prior study proposed to employ a heavy software-based sandboxing layer inside an enclave [22, 47, 55]. Second, as it requires to use operating system services via system calls, the secure interaction with system calls must be supported. Not only such system call requests must be verified to protect the hosting system, but return values must also be checked to prevent Iago attacks against the enclave [26, 41, 79, 86]. Third, tracking the resource utilization of each enclave is important for serverless computing, as both the service developer and cloud provider must agree on the resource usage. However, the current mechanism allows only one-sided accounting, where either an enclave or OS tracks the resource utilization independently [23, 44, 47]. Finally, the current enclaves cause non-negligible performance degradation for verification when loading functions, hindering fast function deployment for changing demands.

To overcome the limitations of the current enclave model, this study proposes a new HW and SW extension of enclave architecture, called *Cloister*. Cloister provides efficient hardware-supported solutions for the four limitations, with relatively minor hardware changes. First, instead of using software-based sandboxing, Cloister proposes a hardware-based sandboxed enclave (*sbx-enclave*), which blocks accesses from enclaves to the untrusted world. By simply extending the pre-existing memory validation mechanism in SGX hardware, an sbx-enclave can not only be protected from the untrusted world but also be prevented from accessing the untrusted context. Such bi-directional isolation enables solid sandboxing support for each sbx-enclave without any extra software layer.

The second mechanism is to provide a hardened interaction between an sbx-enclave and the operating system. The interaction of the sbx-enclave and operating system is forced to go through the monitor enclave to process system calls only if they are valid. The key difference from the prior approaches [22, 47, 55, 76, 89] is that the monitor is isolated both from the sbx-enclave and from the operating system, which provides stronger protection for the system call verification and return value validation. The codes running in the monitor enclave are attested by the sbx-enclave and operating system, providing verified monitoring operations by the two entities.

Third, with the neutral monitor enclave, Cloister can provide a tamper-proof accounting service of system resources such as CPU, memory, file I/Os, and network usage, as both the cloud provider and service developer can trust the monitor enclave. With the support, the resource usage can be reported in a mutually trusted way for

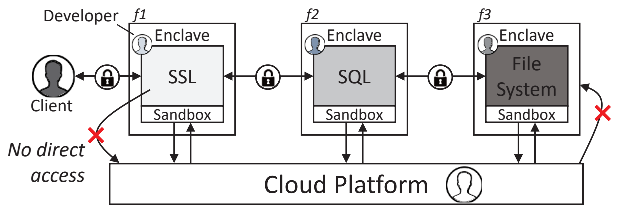

Fig. 1. Overview of trusted serverless computing

both parties. Finally, to accelerate dynamic deployment of functions, Cloister significantly reduces the function launching overheads of the original SGX by parallelizing memory measurement and eliminating redundant operations.

Cloister proposes hardware extensions to the enclave while minimizing required hardware modifications. To show the effectiveness of the new enclave extensions, we use two evaluation methods. Architectural simulation validates that the performance degradation by the bidirectional isolation is negligible. In addition, we port several application scenarios on an emulated SGX runtime with the extended interface. The experimental results show that minor hardware extensions can improve the efficiency and security of serverless applications on clouds. Compared to the prior SW-based sandboxing, Cloister shows up to 44.1% better performance in query servers with multiple functions and 94.3% faster loading with 8x parallelization.

This paper aims to support trusted serverless computing and makes the following new contributions:

- It proposes bi-directional isolation between an enclave and its untrusted environment with a simple extension of the existing memory access control mechanism in SGX. With the hardware extension, it can eliminate an extra SW layer in the prior trusted sandboxes.
- It proposes a hardware-protected monitoring mechanism for handling system call filtering and return value validation.
- Using the monitor enclave, it allows trusted accounting of each enclave resource usage, so that both the service developer and cloud provider can agree on the resource utilization collected by the CPU hardware and monitor enclave.
- To meet the performance requirement of serverless computing, we propose a hardware extension that accelerates enclave loading by reducing measurement overheads.

2 BACKGROUND

2.1 Serverless computing

A serverless application is composed of fine-grained functions which communicate with each other through pre-defined APIs. These functions are event-driven and user-level codes running in an isolated environment. Generally, there are three types of participants in this system: cloud provider, service developers, and clients.

The cloud provider offers the underlying infrastructure including hardware, system software, and runtime to build an application. On top of that, the service developer builds applications by deploying functions so that their clients use them remotely. The developer leverages function modules provided either by the platform or by other developers to simplify the development process. [25, 48, 66]. For better responsiveness and availability, the cloud provider automatically scales the number of functions in applications by dynamically loading and unloading functions. This dynamic nature of serverless computing requires fast function loading.

Serverless computing adopts a pay-as-you-go pricing model in which developers only pay for the amount of resources used by deployed functions. To support the payment model, it is important to have a fine-grained resource accounting system for the dynamic resource consumption used by functions. The cloud provider runs monitoring tools to accurately track resource usage of each function.

Trusted serverless computing: Serverless computing poses both new challenges and opportunities for the security of cloud computing. It decomposes an application into small functions, opening the possibility of compartmentalized execution of functions to avoid the propagation of vulnerabilities across different functions used for an application. However, to exploit the new opportunities from the decomposition, each function must be running on a sandbox, and the bidirectional isolation and secure system services are needed. This study is to improve the security and efficiency of such sandboxed execution required for serverless computing.

Serverless computing needs more complex security models than traditional cloud computing. Three mutually distrusting participants are involved in this system with different requirements. The client should protect their data from the service developer and the cloud provider. The cloud provider must protect their system from the service developer and client with a safe isolation mechanism. In addition, because each function can be implemented by a different developer, they should run in different protection domains.

In addition, billing and monitoring should be protected as well [23, 42, 44, 47], as the service developer and provider have conflicting interests. Since the resource usage measured by an untrusted counter-party is unreliable, it calls for a trustworthy accounting mechanism that both the cloud provider and developer can trust.

To support such protection, recent studies have proposed to run each function in a user-level enclave, and applied additional sandboxing mechanisms to confine the function running in the enclave [55, 69, 76]. The sandbox blocks accesses to the untrusted memory from the enclave to limit the access boundary of an enclave only within its own memory. On top of the protection model, they allow controlled interactions between the function and platform under the supervision of a trusted reference monitor. The monitor is a trusted computing base (TCB) of the application which mediates system calls and APIs.

Figure 1 presents a serverless application consisting of SSL, SQL, and user-level file system. Each function is enclosed with both enclave and sandbox isolating their execution from the platform and any other functions. The functions communicate with each other through pairwise protected channels while their system call interfaces are controlled by the monitor.

2.2 Intel Software Guard Extensions (SGX)

Intel SGX provides a user-level trusted execution environment called an *enclave*. The context of an enclave is protected by the hardware mechanism. The protected memory region of an enclave is created in Enclave Page Cache (EPC). The virtual address range for an enclave should be a single contiguous region called Enclave Linear Address Range (ELRANGE). Part of physical memory, Processor Reserved Memory (PRM), is reserved for SGX and is protected by the hardware memory encryption engine (MEE). PRM contains EPC pages in addition to other security meta-data for SGX. Although EPC pages are in the external DRAM, their confidentiality and integrity are guaranteed under direct physical attacks on DRAM and system interconnection components. The attestation service allows a requester to verify the identity of an enclave and the platform setting where the enclave runs.

The memory isolation for each enclave is achieved during the address translation step for each memory access. A transition between the enclave mode and untrusted mode requires flushing Translation Lookaside Buffers (TLBs). For each TLB miss, the validity of access is verified by the CPU hardware logic. A key internal data structure for verification is Enclave Page Cache Map (EPCM) which is stored in PRM. An EPCM entry has information about a physical page that belongs to the EPC region. It contains the owner's enclave identity and its virtual address in the enclave memory space, in addition to other status information. Even though page tables are still managed and updated by the operating system, the EPCM table is accessible only by the hardware, and the page table entry for EPC can be verified using EPCM. The crucial invariant for the correctness of memory isolation is that *TLB must contain only verified translations.*

SGX controls enclave through a set of instructions. To create an enclave, *ECREATE* creates security metadata of the enclave including SGX Enclave Control Structures (SECS). Then, *EADD* loads the enclave content into the protected memory, and *EEXTEND* measures the loaded EPC to verify its content. Finally, *EINIT* initializes it to be ready for protected execution. The context information of an enclave is stored in its SECS. SECS are allocated in EPC pages for its safety against the malicious operating system. Once an enclave is initialized, even software running in the enclave cannot modify its SECS. SGX writes an enclave digest (*MRENCLAVE*) in SECS which is needed for attestation from *EREPORT*. SGX includes instructions for switching modes between the enclave context and unprotected context: *EENTER* to enter enclave mode, and *EEXIT* to exit enclave mode. In addition, Asynchronous Enclave Exit (AEX) occurs when an enclave generates a fault, or it receives an interrupt. When AEX occurs, the execution context is securely saved in State Save Area (SSA), and registers are sanitized. The saved context will be restored during the next *ERESUME*.

2.3 Sandboxing

Sandboxing is widely adopted for runtime protection against third-party applications, such as web browsers running plugins written by unauthorized developers [9, 27], and testbeds for third-party developers migrating their applications to the production system [3, 8, 15]. An application running in a sandbox must not be allowed to directly access the memory outside of the sandbox. In addition, the application control should never reach beyond the designated sandbox neither directly nor indirectly during its runtime.

Using binary instrumentation, Google Native Client (NaCl) [92] restricts memory accesses from untrusted applications by masking target addresses with memory boundaries before the binary execution. Such software-based isolation needs to execute extra instructions for access validation, adding performance overheads.

In addition to the memory access control, the interaction with the operating system must also be regulated by sandboxing. Although the operating system is protected with privilege separation and system call interfaces, system vulnerabilities via system calls have been continuously reported [4, 11, 13, 14]. The sandbox must provide controlled system functionalities by verifying system calls from the untrusted application. For example, Seccomp-bpf [37] interposes system call requests by filtering system calls with ID and arguments.

2.4 Threat model

Cloister shares the basic threat model and trusted computing base (TCB) of SGX. The SGX-enabled processor package is trusted. Privileged software such as the operating system and hypervisor can be compromised by its vulnerability or any person who obtains the privilege permission. Moreover, attackers can wield direct physical attacks on onboard interconnections and external DRAM. We assume that each function does not fully trust the other functions, even when they are used together to build an application. In our model, the code running in the monitor enclave (5488 LOC) is trusted. The monitor enclave runs on its own enclave, isolated from sbx-enclaves, and it is trusted by the service developer and cloud provider. The integrity of the monitor can be verified using attestation, a key feature of SGX. To establish this monitor, the cloud provider develops the monitor enclave code and release it as open-source. This public approach enables cross-verification by the user community. Additionally, the monitor code can undergo comprehensive functional correctness proof, mirroring the approach of the formally verified operating system, seL4 [59]. The monitor code is much smaller than the operating system, facilitating such verification efforts.

Foreshadow attack [31], side-channel attacks [29, 30, 49, 63, 80, 81, 88, 91], controlled channel attack [67], and availability are not considered in this work. For such attacks, prior patches [19] and protections [21, 40, 67, 70, 75, 77, 78] can be used as orthogonal measures. Cloister does not support resiliency to code reuse attacks [29, 62] and arbitrary API invocation [58].

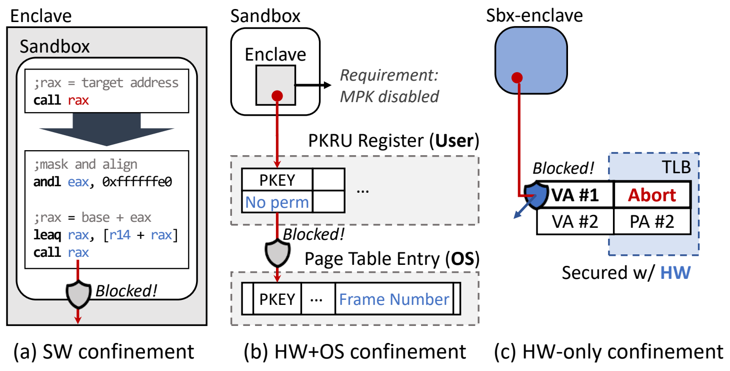

Fig. 2. Three confinement approaches: SW, HW+OS (MPK), and HW-only (Sbx-enclave)

3 MOTIVATION

3.1 Requirement of trusted serverless computing

Supporting trusted serverless computing poses new challenges in the current SGX model. First, TEE not only needs to protect applications itself in an enclave but also must confine accesses from the applications inside enclaves. Second, a function in an enclave often needs to access the system resource via system calls, so it requires a method to control the interaction with OS in both directions. Third, the resource usage of cloud functions must be accurately monitored and tracked for billing in a way that both the cloud provider and service developer can trust. Finally, after applying all of the above protection techniques, there should be no significant performance degradation for loading and running serverless applications under dynamic load changes.

3.2 Challenge 1: Bi-directional Isolation

In trusted serverless computing, an enclave execution must be protected, but it must also be prevented from accessing memory beyond its own EPC region. In the current SGX model, the in-enclave execution is freely allowed to access the rest of its process memory, which is confined only by the operating system. There are two different ways of providing confinement supports for the current SGX enclave model. Figure 2 presents the different approaches.

Figure 2 (a) describes the software-based confinement (SW). This technique is to place an instrumented application binary and sandbox libraries together in an enclave. In this approach, both the sandbox library and application binary must be trusted by application developers, increasing the Trusted Computing Base (TCB) of the enclave.

When vulnerabilities exist in the sandbox library, the application code can exploit the vulnerability to bypass the confinement of the sandbox. We find that many vulnerabilities in software sandboxing have been reported in CVEs (*keyword:Sandbox*)[5] from 2019 to 2022. For instance, attackers can exploit them in two primary ways. First, they can execute unauthorized codes outside an established sandbox, bypassing its defenses [1]. Alternatively, they can intricately design and deploy malicious codes directly within the sandbox, potentially compromising its intended functionality [2, 10].

The software-based confinement incurs performance overheads because every memory access from an enclave has to be verified by instrumentation. A recent study reports that the software-based confinement causes 12.43% slowdown on average, up to 24.89% compared to native execution because it adds 23.52% more instructions [22].

Recent hardware supports for memory confinement such as Intel Memory Protection Keys (MPK) can mitigate the weaknesses of software-only approaches. Figure 2 (b) shows the hardware-assisted approach with MPK,

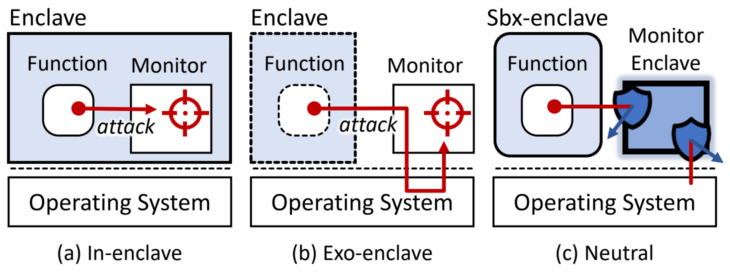

Fig. 3. Three system call monitor approaches: in-enclave, exo-enclave, and neutral enclave

called HW+OS confinement. MPK is designed for user-level confinement, and thus the domain permission can be changed by user-level instructions, which updates the PKRU register defining the permissions. Therefore, the confined application in an enclave must be prohibited from updating the PKRU registers by additional software mechanisms [34, 87]. In addition, since MPK can be nullified by some classes of system calls, MPK-based sandbox must filter those system calls [73]. Furthermore, the MPK and page access permission mechanism required for the HW+OS confinement relies on the security of page tables. However, recent studies showed that page tables can be vulnerable to various attacks based on rowhammer [35, 50, 74, 90].

To overcome the limitations of the current SW and HW+OS confinements, this paper proposes a new HW-only confinement mechanism that eliminates SW from the mechanism. Figure 2 (c) shows our approach of Cloister. In our approach, the confinement extends the current protection mechanism of SGX. With security metadata stored in SECS and EPCM, the integrity of which is protected by the hardware memory protection of SGX, resilient from rowhammer attacks.

3.3 Challenge 2: Secure Interaction with OS

A sandbox enclave is prohibited to access the memory outside of its own EPC, but it should be allowed to issue system call requests if the system call requests can be verified for their safety. Therefore, to provide fully functioning sandbox enclaves, it is necessary to support a safe mechanism to verify system call requests and forward the filtered requests to the operating system. In addition, the returned value from the untrusted operating system needs to be checked.

There are two different approaches to providing system call services to enclave execution: system call emulation and system call delegation. First, the system call emulation approach imports the entire library OS and C standard libraries inside an enclave [26, 28, 76, 86]. However, since the intra-enclave libOS runs in user mode, it still transfers some system calls to kernel (e.g., 42 syscalls among 192 in GrapheneSGX) [65]. In addition, this approach adds the entire software stack within an enclave, increasing 20 KLOC (kilo lines of code) - 1,348 KLOC of TCB [26, 79, 86]. Formal verification for such large TCB with POSIX-compatible interfaces is hard to achieve [39, 59]. Figure 3 (a) shows the in-enclave monitor approach. It assumes that the monitor code can be completely isolated from the function by a software confinement layer. However, the vulnerability in SW confinement may not guarantee the protection of the monitor.

Second, the system call delegation approach relies on the underlying OS itself, thereby reducing TCB drastically [79, 89]. It delegates system calls to the non-enclave mode and performs the system calls. Figure 3 (b) shows the exo-enclave monitor approach. It assumes that the monitor code exists as a process in unprotected environment accessible from the OS kernel. Therefore, this approach is vulnerable to attacks via privilege escalation [12, 13, 43]. In addition, a malicious OS can leak the application secret or break the execution integrity by manipulating return values of system calls [33]. To prevent such an attack known as Iago attack, return values also need to be validated by trusted entity [53, 60].

Service	Invocations	CPU (Second)	Memory (GB-s)	Storage (GB)	Network (GB)
AWS Lambda [25]	✓	✗	✓	✓	✓
Azure Functions [66]	✓	✓	✓	✓	✓
Google Cloud Functions [48]	✓	✓	✓	✓ ‡	✓
IBM Cloud functions [56]	✗	✗	✓	✗	✗

✓: Included ✗: Not included ‡ tmpfs

Table 1. Pricing policies for commercial serverless platforms

Note that the interaction among multiple enclaves and the monitor needs to be considered. Sandboxes using system call delegation have to consider races between sandboxes [46]. Moreover, a prior study [41] observed that Iago attacks can occur across multiple components, thus checking the return value within each enclave individually is not enough to prevent such attacks.

As shown in Figure 3 (c), Cloister takes a different approach. To allow the controlled interaction between the sandbox enclave and operating system, we propose to add a position-neutral *monitor enclave* that can be coupled with one or more sandbox enclaves. In the approach, the monitor is protected both from the sandbox enclave and OS kernel. The monitor can track global states among sandbox enclaves and thus can prevent Iago attacks. In addition, the design helps developers not to add Iago attack protection in every sandbox enclave.

3.4 Challenge 3: Mutually Agreeable Resource Accounting

One of the requirements for the trusted cloud service is tamper-proof resource accounting [23, 24, 47, 57, 84]. For each developer, the system resource usage must be securely tracked and reported. In addition, to satisfy service-level agreement (SLA), the cloud provider must allocate the contracted amount of resources in a verifiable way. Table 1 shows the pricing policies for serverless platforms from major cloud providers. For enclave-based cloud applications, recent studies investigated self-accounting techniques [23, 44, 47, 84]. They are designed to measure the resource usage inside an enclave, which is protected from the cloud provider. The measuring code is regarded as trusted entity and located in the same enclave.

However, the key problem of the current techniques is that it is a one-sided accounting mechanism. Service developers are checking their resource usage within their enclaves. The cloud provider also tracks the resource utilization with the system mechanism. However, how to achieve consensus when developers and cloud providers have different accounting results remains an open problem. As the accounting code is running in an enclave, it can be vulnerable from potential compromise by application codes within the enclave.

To support such tamper-proof accounting which can be trusted by both the cloud provider and developer, it is necessary to track system resource usages by a mutually trusted entity. As a monitor enclave can be isolated from both the developer enclave and OS, it can act as a neutral accountant, recording the utilization of CPU, memory, and I/Os. File and network I/O operations can be tracked with system call interactions, visible to the monitor enclave. However, the monitor enclave cannot directly measure CPU cycles and memory pages used by a sandbox enclave, requiring hardware extensions to record the CPU and memory usages.

3.5 Challenge 4: Fast loading

To build scalable and responsive services, one of the essential requirements is fast function loading. To do so, each function should be lightweight, and its loading process should be simplified. However, for trusted serverless computing, it is necessary to verify whether the function is correctly loaded without manipulation. The hardware computes (measures) the hashed value from the memory contents of the function, and it verifies the loaded function by comparing the measurement to the one made during its compile time.

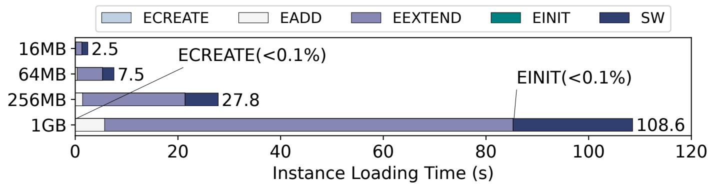

Fig. 4. Enclave loading time breakdown with four different enclave sizes: 16MB, 64MB, 256MB, and 1GB.

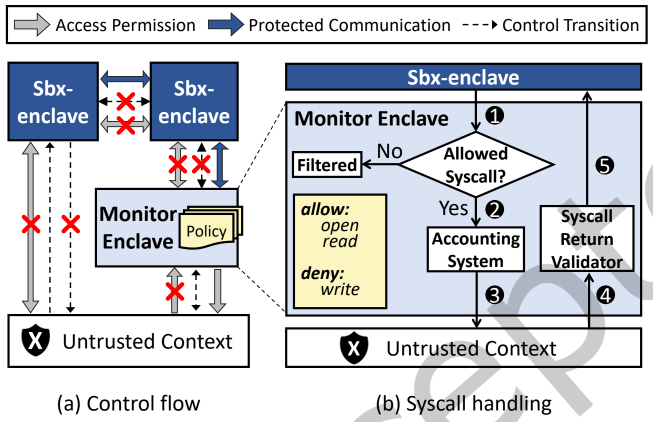

Fig. 5. Cloister architecture

A recent study raised the problem that trusted serverless instance loading is slow [64]. To alleviate the problem, it introduces shareable enclaves. Rather than launching a heavy host function every time, it launches a minimized host function and maps it to an already launched function to share its memory. Similarly, Cloister reduces enclave sizes by decoupling and sharing monitor as a neutral enclave. However, the loading latency still needs to be further improved for newly launched enclaves.

Figure 4 presents the breakdown of enclave loading time when the enclave memory size is 16MB, 64MB, 256MB, or 1GB. As shown in the figure, each loading procedure includes a series of SGX instructions, and other software preparation. It shows *EEXTEND*, which measures each EPC content and relative position with SHA-256, incurs the most significant overhead. On the other hand, *ECREATE*, *EINIT* cause less than 0.1% of the loading overhead. Since *EEXTEND* occupies more than 65% of the loading time on average across enclave sizes, we focus on improving the performance of *EEXTEND*.

4 ARCHITECTURE

4.1 Overview

Figure 5 presents the Cloister's serverless computing model. In Cloister, a sandbox enclave called *sbx-enclave* contains a function after its initialization procedure starting from a normal enclave. Once an sbx-enclave is initialized, it never reverts to the original enclave during its lifetime. The code running in an sbx-enclave is not allowed to read, write, or execute contents outside the sbx-enclave memory. In addition, the control of an sbx-enclave cannot be directly transferred to the non-enclave context. Instead, the sbx-enclave must go through the monitor enclave to interact with the rest of the system. Cloister allows the monitor enclave to communicate with the operating system (OS).

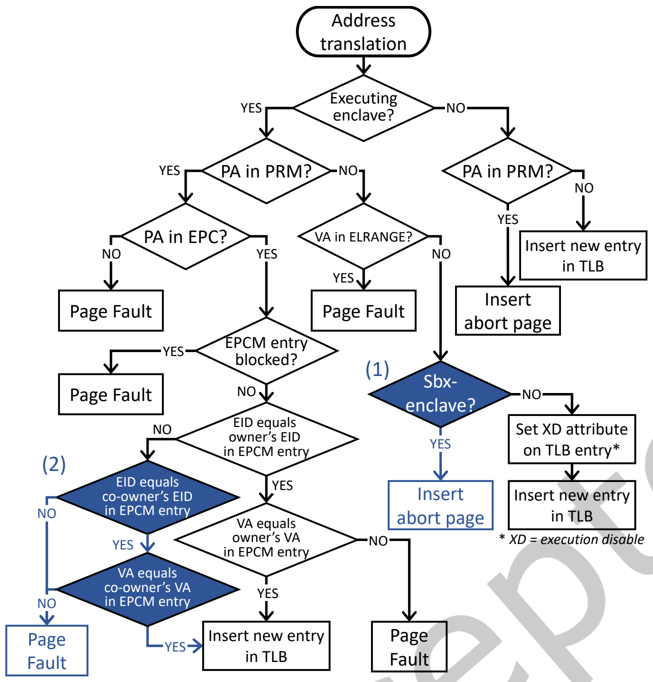

Fig. 6. Access control flow for Cloister. Modification are marked in blue on the original SGX flow [38]

The monitor enclave works as a proxy to communicate with the operating system. To interact with the operating system, an sbx-enclave establishes a secure shared memory channel to the monitor enclave. For communication with other enclaves, the same secure shared memory channel is supported. As shown in Figure 5 (b), to issue a system call, the sbx-enclave must deliver a system call to the monitor, and the monitor enclave executes the system call on behalf of the sbx-enclave. The monitor enclave verifies system calls based on a given profile and validates the return values of system calls to prevent known Iago attacks.

In the Cloister model, a service consists of one or more mutually distrusting functions. Each function is enclosed and protected by an sbx-enclave, and mutually distrusting functions do not reside in the same sbx-enclave. To make the service scalable, both the sbx-enclave and monitor can have multiple threads in the protection boundary. A common use scenario for mapping the monitor to sbx-enclaves is that a cloud application by a service developer can create a monitor enclave and the sbx-enclaves constituting the application can share the monitor enclave, as they have the shared security goal and accounting records. The monitor enclave can also function as the trusted reporter of the resource utilization records of the sbx-enclaves connected to it. As the monitor enclave is trusted by both its developers and the cloud provider, its resource usage reports are mutually trusted by both parties.

4.2 Memory Protection for Sbx-enclave

Access validation: Cloister augments SGX memory protection features to enable bi-directional memory protection. It protects the non-enclave memory context by preventing memory translation from an sbx-enclave. Figure 6 is the hardware flowchart of Cloister's address translation. **(1)** in the figure indicates the additional memory protection added for Cloister. Cloister verifies whether an enclave is an sbx-enclave by checking the flag set in SECS which is immutable by any software after initialization. When an enclave is not in the sbx-enclave mode, memory accesses to a non-ELRANGE virtual address are allowed. Thus, Cloister inserts a new entry to the TLB without execution permission in the same way as the original SGX. When the enclave is in the sbx-enclave mode, the sandboxed code must not be allowed to access the outside memory. Therefore, Cloister inserts an abort page to cause a failure in resolving the non-ELRANGE virtual address to a physical address. As shown in the figure, the extra access control for sbx-enclaves does not require any significant hardware changes; Cloister adds minor extra condition checks while handling a TLB miss.

Control transition: To execute the codes in non-enclave locations, an enclave needs to transfer its control to the outside code. By calling *ocall*, an enclave performs a control transfer from the enclave to the non-enclave context. During an ocall, a normal enclave saves its state in the protected memory, cleanses all internal CPU states to prevent security leaks, and switches its mode into the non-enclave mode with *EEXIT*. However, *EEXIT* can be exploited to jump to an arbitrary non-enclave memory location by setting the *RCX* register to the destination.

Unlike normal enclaves, Cloister isolates an sbx-enclave by disabling the *EEXIT* instruction. The hardware modification for *EEXIT* is straightforward since the only modification is raising an exception in the sbx-enclave mode. However, AEX is still allowed even for the sbx-enclave because the operating system must handle exceptions such as page faults or interrupts. Cloister switches its execution mode to handle the exit events. The event is handled by designated hardware exception handlers in the kernel. During AEX, it erases any context (secrets) that may exist in the execution state [38]. Therefore, the software in an sbx-enclave cannot exploit AEX for escaping the sandbox. Cloister does not modify the execution flow of AEX from the SGX.

Sharing EPC: The current software-based encryption on the shared untrusted page is not only inefficient but also must make a non-enclave page accessible by the sbx-enclave, which is inconsistent with the bi-directional enclosure property. Cloister adopts the shared channel design using hardware supports from the recent secure channel proposals for RISC-V TEEs [45, 93]. Cloister extends each EPCM entry to have mapping information of the co-owner which includes SECS address, VA mapping, and permissions. Figure 6 **(2)** shows the hardware extension for the ownership checking. When a memory access occurs to EPC, Cloister checks the corresponding EPCM entry and verifies the ownership. The EPCM entry has at most one co-owner, thus only two different enclave contexts can access the EPC. Based on the shared EPC, Cloister support both synchronous and asynchronous communication between enclaves.

Advantages: Cloister does not require any extra SW layers or compiler-based validations for the confinement, minimizing the performance overhead while providing hardware-enforced confinement. Moreover, Cloister can support self-modifying code and JIT compilation in sbx-enclaves. Cloister keeps the critical meta-data in the secure memory region (PRM). Therefore, the meta-data is protected from OS and DRAM attacks including rowhammer because any bit flips in PRM are detected by the integrity validation of MEE.

4.3 Sandbox Monitor

An application using one or multiple sbx-enclaves needs to have a monitor enclave. The monitor enclave is multi-threaded and it allocates a dedicated monitor thread to each sbx-enclave in the application. During runtime, the individual state of each sbx-enclave is tracked by its monitor thread. Both SGX and Cloister do not allow invoking system calls within an enclave. For system calls, an sbx-enclave calls the monitor enclave with the same interface as ocall in SGX. Note that the monitor enclave is a conventional enclave, and thus it can jump to the system call function in the untrusted region.

System call handling: As shown in Figure 7 (a), the monitor enclave executes a software reference monitor which verifies system calls and returns values. For legitimate ones, the monitor enclave handles the system calls on behalf of sbx-enclaves. If necessary, the monitor can request system calls to the kernel or can use SGX-provided C standard libraries [16]. However, there are system calls that the sbx-enclave thread should execute by itself

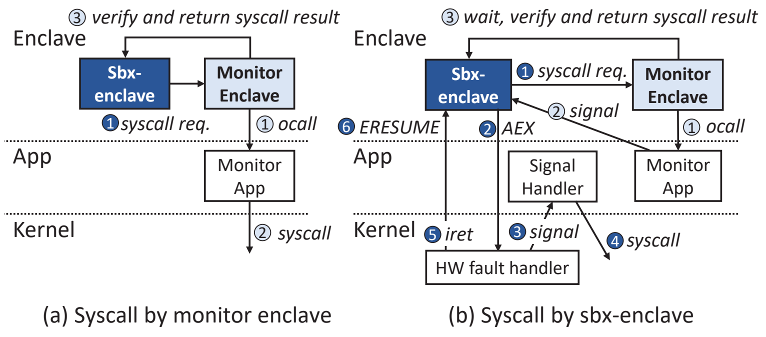

Fig. 7. Two system call flows in Cloister

(e.g., *sleep*, *exit*, *getpid*). Figure 7 (b) shows how Cloister handles those system calls with signals. When such a system call is requested, the monitor sends *S*IGUSR1 signal to the target sbx-enclave. It triggers AEX and invokes a custom signal handler. The handler executes the requested system call, and returns the result through the monitor. The design enables the monitor to let the sbx-enclave run the system call by itself securely without ocall.

Development: In Cloister model, The monitor enclave (and accounting program) is developed by the cloud provider and open-sourced by them. Cloister keeps its design simple to reduce risks. Developers need to inform the cloud provider the APIs to use in its policy file. The cloud provider must trust the accounting in the monitor enclave because they are utilizing their own code. As the code is open-sourced, the developer community cross-verifies the code. For each service launch, the service developer performs remote attestation on the monitor enclave to ensure its integrity and security.

The monitor verifies system calls from sbx-enclaves by its policy definitions. It reads a policy definition file for each sbx-enclave which is mutually agreed and shared in advance by the function developer and the cloud provider. Furthermore, if needed, different enclaves can share the same monitor enclave policy templates. These templates ought to be supplied by cloud providers and filled by developers. To verify that the policy file is correctly loaded, both an sbx-enclave and OS can query the monitor enclave to obtain the digest of the file. The sbx-enclave checks whether the digest value matches with its own policy digest to make sure that the file is not manipulated by the OS. The monitor enclave opens an up-call interface only for serving the digest query from the OS. This is similar to the attestation service for SGX to verify an enclave.

Monitor as mediator: Similar to seccomp-bpf [37], the monitor enclave filters each system call with the system call ID and its arguments based on fine-grained privileges to system resources through an allowlist and a denylist. To speed up the system call filtering, the monitor also adopts an action-based policy. If an action specifies KILL, the monitor sends a request to the kernel to terminate the sbx-enclave. On TRAP, the monitor runs a customized logic (e.g., sending a message).

When a system call returns, the monitor enclave verifies the return value from the kernel to prevent Iago attacks. Similar to Panoply [79], Cloister checks the types of return values. System calls have three types of return values: 0 or error, integer, and structure mostly with integer fields. Cloister's monitor can examine the range of return values for system calls, identifying if they are True or False or if they lie within a certain integer range.

Cloister checks whether *futex*, *locks*, and *semaphore* are not shared between the sbx-enclave and untrusted world. For a system call that returns a descriptor or reference (e.g., *open*, *socket*), the monitor keeps it in its memory so that the returned descriptor is not substituted or reused. In addition, Cloister is resilient to pointer misuses since the reference is not accessible by an sbx-enclave.

The monitor enclave mediates the secure channel establishment according to the policy definition. When an sbx-enclave tries to build a secure channel with the specified sbx-enclave on the policy, the monitor allows the connection. Otherwise, it refuses the channel establishment.

4.4 Trusted Accounting

When the cloud platform and the sbx-enclave perform accounting separately, problems arise when the two values differ. To avoid such a conflicting accounting problem, in Cloister, a neutral accounting monitor is protected by the hardware and trusted by both the cloud provider and service developer.

The monitor enclave securely collects the resource usage of the sbx-enclaves it is serving and can provide the service developer and cloud provider with an authenticated report on the usage. Since the monitor is trusted by both parties, its report provides a mutually agreeable accounting, unlike the current separate accounting either by the developer or by the cloud provider. The monitor enclave can collect the file and network usage directly as all system calls pass through the monitor. However, it requires hardware extension for CPU and memory usage.

CPU and memory usage: The hardware extension will collect the information and write logs for the monitor enclave. The monitor enclave reads the logs for the CPU execution time or the number of pages used by an sbx-enclave. The log area is allocated as a shared page between an sbx-enclave and its monitor, similar to the communication channel setup. One difference between the log area and communication channel page is that the log page is only writable by the hardware. The sbx-enclave and monitor enclave can only read the data after it is initialized. Therefore, the monitor and sbx-enclave can read the same log, but they cannot change that. The required hardware modifications for this purpose are minimal. The existing SGX already provides the capability to offer multiple types of EPC with different read/write permissions. For CPU logs, we only need to add a new page type that has read permissions but not write permissions. Therefore, when the logs are stored in memory, no additional encryption is necessary as they are already securely stored with MEE and access control. On the other hand, when logs are saved as files, Cloister employs encryption through the Protected Filesystem to ensure their security and prevent tampering during storage.

To track the usage of CPU cycles, the CPU hardware will trace the execution cycle between the start and end of the execution period for an sbx-enclave. The starting time is measured at the execution of *EENTER* and *ERESUME*, and the end time is measured at *AEX* and *EEXIT*. The execution cycle is accumulated to the CPU log when the current execution period ends at *AEX* or *EEXIT*. One adjustment of the measured CPU cycle is the power state. The hardware saves power state along with the CPU cycle. Note that manipulating Time Stamp Counter (TSC), and CPU power state cannot be performed during enclave execution, since those need kernel privilege [17].

To track memory utilization, Cloister hardware updates the log for adding EPC (*ECREATE*, *EADD*) and removing EPC (*EREMOVE*). When the memory is augmented (*EAUG*) or deallocated (*EMODT*) during the lifetime of an enclave as supported by SGX2, the acceptance event (*EACCEPT*, *EACCEPTCOPY*) will also update the log. In addition, Cloister tracks how many EPC pages reside in PRM. On EPC writeback (*EWB*), it decreases the number, while EPC loading (*ELDB*, *ELDU*) increases the number. To avoid races, hardware logging for the log structure is serialized. Note that an sbx-enclave never executes instructions or accesses memory outside of the enclave as a sandbox, although minor untrusted housekeeping codes can be executed for signal handling. Therefore, its non-enclave resource utilization is not tracked by the hardware.

Network and file I/Os: To measure I/O usage of each sbx-enclave, the monitor enclave tracks the number of invocations of the I/O APIs (e.g., *open*, *connect*). In addition, it counts the amount of data passed for APIs (e.g., *read*, *write*, *fstat*). The monitor enclave can record tamper-proof evidence for network and file usage because all accesses (system calls) to the resources must pass through the monitor enclave. Therefore, the monitor enclave can log resource requests from each sbx-enclave, and neither the sbx-enclave nor the host OS can modify the log contents.

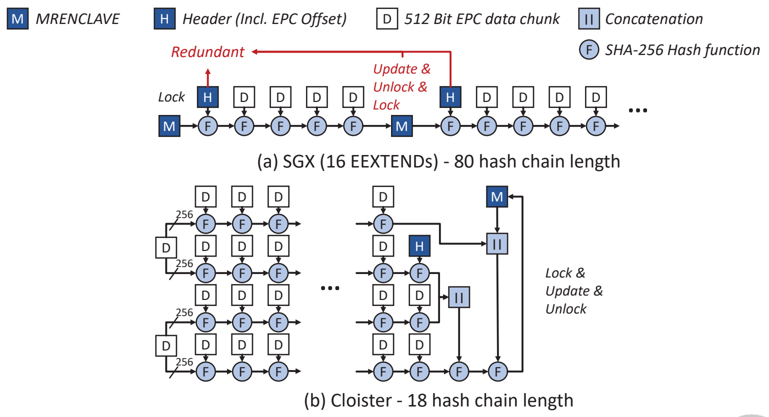

Fig. 8. Measuring an EPC page: SGX (a) vs Cloister (b)

4.5 Hardware Extension for Fast Loading

Measuring EPCs accounts for the majority of loading times for launching a new function instance. Figure 8 (a) shows how SGX measures an EPC. SGX invokes *EEXTEND* 16 times for each EPC and each *EEXTEND* performs SHA-256 hash function 5 times. With the hashing, *EEXTNED* measures the content and relative position of the EPC. For each time, it accumulates the result on *MRENCLAVE* field in SECS. The measuring process is serialized incurring a significant slowdown in enclave loading. Furthermore, *EEXTEND* redundantly acquires and holds a lock of SECS until it updates *MRENCLAVE*. To alleviate the overhead, Cloister suggests two optimizations, eliminating redundant operations and parallelizing EPC measurement. Figure 8 (b) describes the optimized model of Cloister with 4x parallelization.

First, Cloister eliminates repetitive hash operations for the *MRENCLAVE* and header values by performing them in the last stage. It not only reduces the number of lock acquisitions and the holding time but also gets rid of redundant inputs. The header includes location information for the EPC, and has the same content among consecutive 16 *EEXTEND*s. The optimization reduces 18.75% of hash calculation and about 33% of input data without a loss of information in the content and position. In this model, the measurement is performed atomically at the 4k page level, rather than being atomic per 256 bytes. Similar to the original SGX, this model can still be interrupted before each 4k measurement.

Second, parallelization can be done by locating extra hash modules. The more hash modules, the higher level of parallelism can be achieved. Once Cloister measures an EPC, it splits an EPC, hashes the content in parallel, and accumulates the results on *MRENCLAVE*. The parallelization does not compromise security because it preserves the entire contents and order information in the *MRENCLAVE*. In a 4x parallelization with the optimizations, Cloister can reduce 77.5% of the hash chain length from the original flow. As with the original *EEXTEND*, controlling the hash modules can be performed with microcode. Also, recent study shows Intel's SHA-256 module takes 4916.7μm^2 area in 14nm ASIC [94]. Using a faster hash function (e.g., *keccack*), or measuring multiple pages at once may accelerate this process.

5 EVALUATION

We use two methods to evaluate the performance of Cloister. The first method is to use a microarchitecture simulator to assess the performance impact of the access validation mechanism of sbx-enclaves. The second method uses an emulated SGX system to evaluate full application scenarios.

Benchmark (1000 iterations / sec)	Hardware mode	Emulation mode
NBench (geomean) [20]	6.0	6.4
SGX ecall / ocall (switchless)	2283.8 / 3748.5	3110.3 / 3783.7
Cloister inter-enclave call	-	4930.5

Table 2. Performance comparison for the real SGX hardware and the emulated system

5.1 Microarchitecture Simulation

To measure the overhead from microarchitecture changes affecting TLB miss handling with Cloister, we model the confinement mechanism in Zsim [72] with DRAMSIM2 [71]. This simulation-based method evaluates the detailed architectural overheads caused by the bidirectional protection, which complement the full-system emulated SGX evaluation framework which will be discussed in Section 5.2.

Modeled environment: We model TLBs, Paging Structure Cache [18], page tables, and EPCM in ZSim. The simulated CPU is a 4-way out-of-order execution core with 32KB L1 data and instruction caches and 256KB L2 cache. 8 cores share a 16MB L3 cache. L1 data and instruction TLBs have 64 entries, and L2 TLB has 512 entries. We use a smaller L2 TLB capacity than the latest Intel processor with 1.5K entries, to stress TLBs with our benchmarks and to find any performance degradation by the additional mechanism by Cloister. The external memory is simulated with DRAMSIM2 with the DDR3-1600 configuration.

Simulation: We simulate the modified address translation by Cloister as shown in Figure 6. When a TLB miss occurs, CPU brings security metadata for SGX into the caches [38]. We assume that an EPCM entry fits in a single cache line after adopting Cloister (350bits [38] + 100bits (Cloister)). Memory contention from data and metadata accesses are also considered. In addition, to simulate the effect of communication channel, we set the first few consecutively accessed pages as the communication buffer and measure the overheads of the TLB miss handler's extra checks and EPCM entry reads. Since Cloister does not change MEE and EPC page swapping from SGX, we added fixed latencies (40-80, 40K cycles) for each event as described in the prior work [54, 82].

Benchmarks: We select memory-intensive benchmarks from SPEC2006 and Biobench, and run 100 Million instructions for each simulation point. Simulation points are selected with Simpoint [52] to simulate the representative periods. We also run Redis with its own benchmark for 10,000 queries (`redis-benchmark -q -n 10000`). We assume that the entire application is running inside an sbx-enclave.

Results: The results demonstrate that Cloister introduces minimal overhead across all workloads, with the impact being less than 0.1%. In other words, additional check for each L2 TLB miss has negligible performance implications.

5.2 Emulation-based Full-system Evaluation

Environment: We use the simulation mode in Intel SGX Driver and SDK 2.6/2.2 to emulate our new instructions and software model. The simulation mode supports SGX APIs, trusted libraries, and emulation for SGX instructions [7]. We evaluate Cloister in servers consisting of Intel CPU i7-7700, 64GB DDR4 DRAM with Linux kernel 5.4.0. There are 459 LOC modifications made on the SDK and driver. Table 2 shows the performance comparison between the real hardware SGX mode and emulation mode. For a fair comparison between Cloister and the current SGX, in the rest of this section, all experiments were run in the emulation mode.

Instruction emulation: To emulate sbx-enclaves, we modify the emulation code in SGX SDK (*EINIT*, *EEXIT*, *EREPORT*). In addition, we add three new instructions (*ESADD*, *ESACCEPT*, *EUNSHARE*) to support the protected communication channel. When the new instructions update the mapping information in EPCM, Cloister sends

We measure the SGX's extra TLB miss penalty from SGX-enabled machine (<4.7 cycles) which is an additional latency for supporting SGX access validation on top of conventional TLB miss handling.

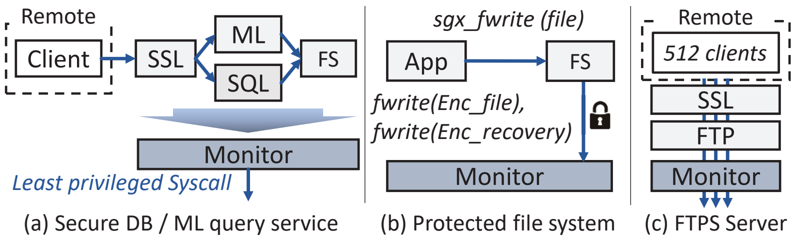

Fig. 9. Evaluation scenarios with serverless computing

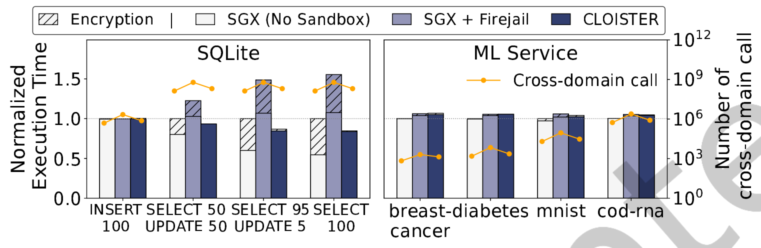

Fig. 10. Performance comparison for serverless query servers

an ioctl to the SGX driver and flushes TLB for the new mapping. The emulated overhead of *EEXTEND* has been set based on the measured latency with the driver (7.2K Cycles).

Communication: We modified *Edger8r* in SDK to generate inter-enclave call APIs from the Enclave Defined Language format. The communication APIs are similar to ecall and ocall, but all the arguments are secured. The generated APIs perform type and boundary checking for each parameter. Since sending a signal to enclave always incurs AEX, CLOISTER prototype uses spinlock for synchronization, which is used in the Switchless Call by Intel SGX [83]. The enclave enters the sleep state after 20,000 *pauses* in the *loop* to reduce wasted CPU cycles [83].

Signal handler: To allow the untrusted thread of an sbx-enclave to handle system calls by itself, we attach a custom signal handler to each sbx-enclave before its control initially enters the sbx-enclave. On a system call request, the monitor enclave writes required arguments to the buffer shared with the sbx-enclave's untrusted thread and sends *SIGUSR1*. The signal causes AEX to the sbx-enclave. After the signal handler executes the system call, it returns the result to the shared buffer.

5.3 Application Performance

To evaluate CLOISTER with serverless computing scenarios, we build secure DB and ML query services. Figure 9 (a) describes the system; each service consists of multiple functions which are isolated in sbx-enclaves. The functions are OpenSSL, Protected FS, SQLite, and LibSVM. The monitor enclave exposes the least privileged system call interfaces to each function. Whenever a client sends security-sensitive data to DB/ML services via SSL, SQLite and LibSVM ask the Protected FS to store or load the data. Protected FS has its own encryption key which is not accessible from other functions. In this experiment, a client sends 10K queries from YCSB [36] for DB, and various training sets from LibSVM [32] for ML.

Figure 10 shows the normalized execution times of the two serverless services, DB (SQLite) and ML (ML Service), over SGX without sandboxing (SGX). The figure also shows the number of required cross-domain calls per query. The cross-domain calls include ecall/ocall, inter-enclave call, and IPC. For the baseline (`SGX+Firejail`), we use Firejail [6] along with SGX. Firejail leverages Linux namespace and seccomp [37] for system call interposition. In SGX and `SGX+Firejail`, communication among enclaves uses software encryption (AES128-GCM) on untrusted

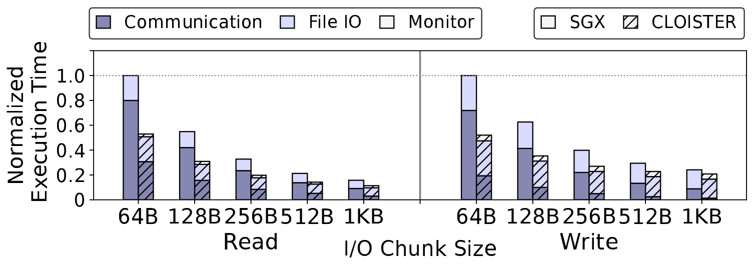

Fig. 11. Execution time breakdown in the file I/O scenario with chunk sizes from 64B to 1KB

memory, while Cloister relies on the hardware-protected shared EPC. In `SGX+Firejail` and `CLOISTER`, every system call must be filtered by the monitor.

As a result, for I/O-intensive SQLite, `CLOISTER` shows 8.9% speedup over `SGX` and 44.1% speedup over `SGX+Firejail` on average. The most significant overhead in `SGX+Firejail` is from additional communication of process-based sandboxing. Since the approach requires a process for each sandbox thread, it needs one ecall, one ocall, and two IPCs for each inter-enclave call [89]. However, Cloister invokes 49.7% less cross-domain call over `SGX+Firejail` by leveraging a pairwise connection. Furthermore, since SGX MEE performs hardware encryption only when data leave on-chip last level cache, Cloister can skip most of the encryption for frequent shared EPC accesses. For ML service, three systems show similar performance (<5%) because the compute-intensive workload does not incur significant communications.

5.4 Protected File System as a Function

To evaluate the communication overhead between an sbx-enclave and the monitor enclave, we run the case study with a protected file system (FS) as depicted in Figure 9 (b). We deploy a function that runs a protected FS which provides integrity and confidentiality protection of files. The application uses the protected FS to secure file I/Os.

Figure 11 presents normalized execution times with various chunk sizes. Each execution time is normalized to when the chunk size is 64B of `SGX`. To isolate the performance cost of Cloister without the effect of long disk I/Os, we perform file I/Os on the mounted tmpfs (DRAM backend). We break down the execution time into the followings. `File I/O` indicates the time taken for file APIs, and `Communication` shows the execution time for all communications between functions including message serialization and encryption. `Monitor` is the overhead of the monitor. Cloister's HW-based communication effectively saves the communication cost, amortizing 8.4% of monitoring overhead. As a result, `CLOISTER` shows up to 1.38-1.89× faster in `read`, 1.16-1.92× faster in `write` compared to `SGX` when the chunk size is set between 64B-1KB. Fine-grained file I/O operations cause more frequent inter-enclave calls between functions, thus it amplifies the communication cost.

5.5 The Effect of Confinement

We evaluate the efficiency of hardware-based confinement by Cloister. As the baseline, we use Chancel [22], a software-based binary instrumentation approach with SGX for bi-directional isolation. For this evaluation, we run `NBench` [20] which consists of ten benchmarks exposing CPU and memory capabilities. Figure 12 shows the execution times normalized to SGX runs without any sandboxing. In `NBench`, Chancel shows 12.3% performance degradations on average over the SGX runs without sandboxing. The overhead is from Chancel's binary instrumentation which adds additional instructions (+23.5%). However, Cloister runs `NBench` within an efficient hardware-protected confinement and does not cause any performance degradation.

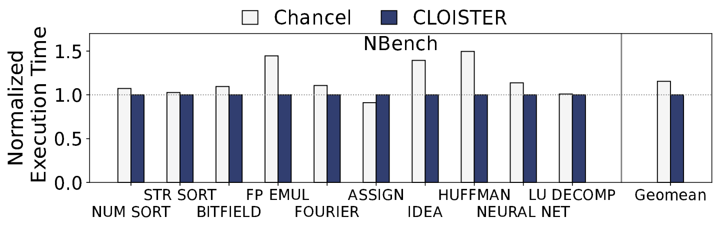

Fig. 12. Confinement efficiency compared to Chancel: execution times with NBench normalized to SGX runs without sandboxing

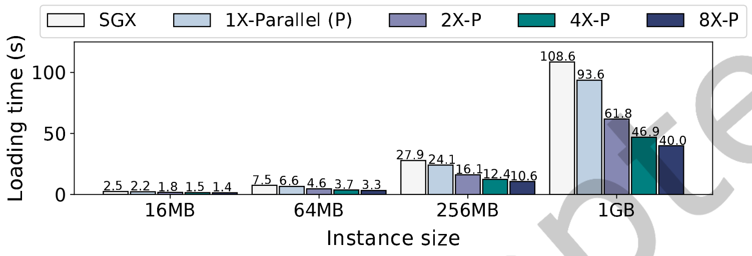

Fig. 13. Cloister instance loading times compared to SGX

5.6 The Effect of Fast Enclave Loading

The enclave loading time is important for nimble deployments of serverless services on clouds. Figure 13 shows the loading speed of Cloister in various enclave sizes. Compared to the baseline SGX, the other configurations indicate different set-ups of loading optimizations in Cloister. For example, 8X-P represents the simulated loading time with 8 SHA-256 modules, assuming that *EEXTEND* takes time proportional to its hash operations. Compared to SGX, Cloister shows 1.73, 2.27, and 2.64 times faster average loading speed with 2X-P, 4X-P, and 8X-P. In addition, even with 1X-P, Cloister shows 15.7% better performance over SGX on average, since Cloister eliminates redundant hashing for *MRENCLAVE* and header, as described in Section 4.5.

We also measure the loading times of Cloister and Chancel with NBench. Cloister is 7.5-59.8% faster over the SGX with 1-8x parallelization including the initial channel establishment. On the other hand, Chancel is 28.5% slower than SGX due to its software binary verification after loading.

5.7 Tamper-proof Accounting System

This section describes the tamper-proof accounting system scenario with FTPS server as shown in Figure 9 (c). To show the effectiveness of the accounting capability, we set up an FTPS (FTP-SSL) server running in Cloister. This configuration is similar to the AWS Transfer Family setup where a server initiates a Lambda function equipped with a specialized file-processing logic. Such a logic can encompass operations like encrypting files, scanning for malware, or cross-checking file types. Furthermore, it underscores that Cloister can tally various system call metrics, including file I/O accesses, network requests, and memory usages.

For the run, 512 clients are running, and each client sends a request for a 1MB file to FTP server with SSL protocol. Handling the requests, the monitor enclave logs per-request resource consumption in its secure memory. Figure 14 shows the latency distribution of the requests with three systems: SGX, CLOISTER, and CLOISTER + Accounting. The median latency of CLOISTER is only 2% slower than SGX. CLOISTER + Accounting shows

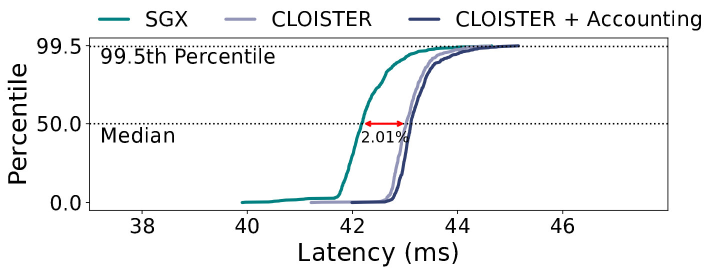

Fig. 14. Latency distribution of FTPS requests

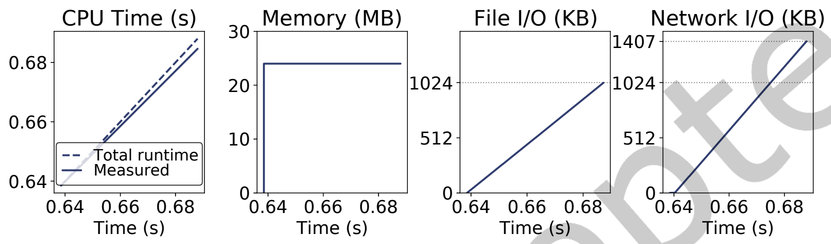

Fig. 15. Accounting results for handling an FTPS request

negligible overhead in median and tail latencies. This implies that trusted secure accounting can be achieved without a large overhead.

Figure 15 shows the accounting results of a single request of the client. The measured CPU time is similar to the total runtime measured by *gettimeofday()*, but with a 6.87% error. This is because the accounting system of Cloister solely counts the sbx-enclave's uptime. The memory graph shows that loading a 24MB FTPS server takes 0.64s and the instance size is not changed. The file and network I/O graphs show the behavior of the FTPS server. Once the server reads a requested file name from the network, it reads the file and sends the contents to the client through the network. We can observe the server sends more data than the actual file contents due to OpenSSL's encryption and metadata.

5.8 The Effect of Cloister

In this section, we provide a summary of the impact of the Cloister in comparison to the identical configurations outlined in the preceding sections (see Sections 5.3, 5.4, 5.5, 5.6, 5.7). Figure 16 presents the combined results across all scenarios, utilizing three setups: `SGX (No Sandbox)`, `SGX+Firejail`, and `Cloister`.

The execution times are normalized to the `SGX` execution time. Each execution times are broken down to four distinct categories: `Loading`, `Execution`, `Encryption`, and `Monitor`. The software organizations of the `DB Service`, `ML Service`, and `FTPS Server` are depicted in Figure 9. Notably, for `NBench`, a solitary instance operates within the sandbox, without any interconnected instance.

We configure `Cloister` with 8X parallel hashing, and each enclave instance is allocated with a memory size of 256MB. In the scenarios characterized by computation-intensive workloads, such as `ML Service` and `NBench`, the incurred costs of encryption and sandboxing remain marginal. Conversely, in the I/O-intensive workloads, such as the `DB Service` and `FTPS Server`, `SGX` experiences encryption-related overheads of 17.8% and 8.4%,

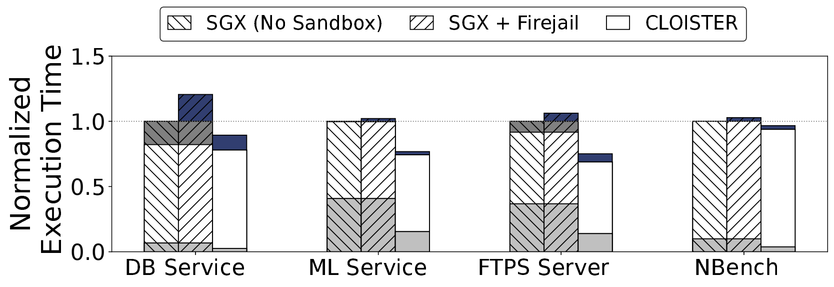

Fig. 16. Performance comparison for all scenarios

respectively, in terms of execution time. Furthermore, the `SGX+Firejail` configuration introduces overheads of 20.6% and 6.3% for sandboxing relative to SGX, due to frequent system calls from I/O operations.

One of the benefits of Cloister is its utilization of hardware-based encryption, which eliminates the need for extra costs in communication between instances and the sandbox monitor. The figure shows that `Cloister` consistently loads 2.64 times faster than both SGX and `SGX+Firejail` setups showcasing the loading acceleration effect by Cloister.

In summary, `Cloister` demonstrates a 3.3% to 24.9% faster instance lifecycle than SGX owing to its loading acceleration, and a 5.9% to 29.3% faster instance lifecycle over `SGX+Firejail`. Most importantly, Cloister provides the hardware hardened two-way isolation and mutually trusted monitor with secure accounting in this setup. These outcomes not only demonstrate the effectiveness of Cloister but also indicate its enhanced security, particularly within the context of trusted serverless computing.

6 RELATED WORK

Sandbox enclave: There have been several prior studies for sandboxing within an enclave. Ryoan decomposes a cloud application into distributed enclaves with the software sandboxing and SW-encrypted channels [55]. Chancel proposes a multi-client software fault isolation technique through binary instrumentation [22]. Occlum leverages Intel MPX for multi-domain software fault isolation within an enclave [76]. SGXLock [34] confines an enclave instance with MPK and make PKRU unchangeable within the enclave. Unlike the prior work, Cloister does not increase software TCB within a sandboxed enclave and it does not prohibit the use of hardware features to enable bidirectional protection.

Trustworthy accounting: Recent studies investigated the need for trustworthy accounting on clouds. HRA first proposed the hardware-based trusted resource accounting for virtual machines when the hypervisor is untrusted [57]. AccTEE measures CPU usage by counting executed instructions [47]. For I/O usage, AccTEE accumulates bytes in and out of the enclave through I/O functions. S-FaaS runs a dedicated thread executing a for-loop to measure the CPU time [23]. To detect AEX, it uses Intel TSX technology in the thread. T-counter measures CPU usage with static analysis and instrumented binary [44]. T-lease uses Time Stamp Counter (TSC) to measure the execution time of an enclave [84]. To protect against TSC manipulation, it calibrates the clock with an attacker-uncontrollerable instruction, *RDRAND*. A major difference between Cloister to the prior work is Cloister does not measure resource usage inside the application enclave. To achieve mutual agreement between the OS and application, Cloister proposes a neutral accounting model with the monitor enclave.

Enclave startup optimization: For nimble instance loading, Clemmys leverages dynamic memory management of SGX2 to skip *EADD* during the startup of an enclave [85]. Plugin Enclave and Reusable Enclave propose reuse-based fast instance loading techniques [64, 95]. By remapping already loaded plugin enclaves, a host application can be loaded faster than cold launches. Unlike the prior studies, CLOISTER proposes hardware acceleration for instance measurement.

TEE communication: To allow efficient communication, recent studies proposed hardware-protected shared memory for RISC-V enclaves [45, 93]. The new communication architectures allow improved performance while preventing intervention from attackers outside. They enable protection from race attacks (e.g., TOCTOU) through ownership transfers or exclusive memory locks. On the other hand, Nested Enclave introduces a shared enclave among enclaves called outer enclave to provide trusted shared memory [68].

7 CONCLUSION

This paper explored a new enclave extension model, CLOISTER, to support hardware-based trusted serverless computing. CLOISTER proposed hardware-hardened sandboxing with *sbx-enclave*, protected OS interaction with *monitor enclave*, trusted accounting with hardware logging, and accelerated enclave loading with parallelization. The design not only reduces TCB within the protection boundary but also resolves security concerns that current approaches have. CLOISTER minimizes hardware modification by leveraging existing hardware features of SGX.

REFERENCES

[1] [n. d.]. An app may be able to execute arbitrary code out of its sandbox or with certain elevated privileges. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-33097. Mar, 2023.

[2] [n. d.]. An attacker with JavaScript execution may be able to execute arbitrary code. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-33097. Aug, 2021.

[3] [n. d.]. Choosing an App Engine environment. https://cloud.google.com/appengine/docs/the-appengine-environments.

[4] [n. d.]. Comodo Antivirus versions up to 12.0.0.6810 are vulnerable to Local Privilege Escalation due to CmdAgent's handling of COM clients. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-3969. Jan, 2019.

[5] [n. d.]. CVE lists. https://cve.mitre.org/.

[6] [n. d.]. Firejail. https://firejail.wordpress.com/.

[7] [n. d.]. How to Run Intel Software Guard Extensions' Simulation Mode. https://software.intel.com/content/www/us/en/develop/blogs/usage-of-simulation-mode-in-sgx-enhanced-application.html.

[8] [n. d.]. Instructions for RVS Sandbox Environment. https://developer.amazon.com/docs/in-app-purchasing/iap-rvs-setup-sandbox.html.

[9] [n. d.]. Mozilla Security/Sandbox. https://wiki.mozilla.org/Security/Sandbox.

[10] [n. d.]. Node.js custom inspect function allows attackers to escape the sandbox and run arbitrary code.. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-33097. Jul, 2023.

[11] [n. d.]. Possible seccomp bypass due to seccomp policies that allow the use of ptrace. https://nvd.nist.gov/vuln/detail/CVE-2019-2054. May, 2019.

[12] [n. d.]. Process-injection: Ptrace System Calls. https://attack.mitre.org/techniques/T1055/008/. Jan, 2020.

[13] [n. d.]. Setting the environment occurs across a privilege boundary from Bash execution, aka "ShellShock". https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271. Sep, 2014.

[14] [n. d.]. Windows Kernel Local Elevation of Privilege Vulnerability. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-17087. August, 2020.

[15] 2013. Building Your Dev and Test Sandbox with Windows Azure Infrastructure Services. https://azure.microsoft.com/ko-kr/resources/videos/build2013-dev-test-sandbox-with-windows-azure-infrastructure-services/.

[16] 2016. Intel(R) Software Guard Extensions SDK Developer Reference for Linux* OS.

[17] 2016. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3C: System Programming Guide, Part 3.

[18] 2019. Intel 64 and IA-32 architectures software developer's manual, Volume 3. (2019).

[19] 2021. Affected Processors: Transient Execution Attacks & Related Security Issues by CPU. https://software.intel.com/security-software-guidance/processors-affected-transient-execution-attack-mitigation-product-cpu-model.

[20] 2022. SGX-NBench. https://github.com/utds3lab/sgx-nbench.

[21] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. 2019. Obfuscuro: A Commodity Obfuscation Engine on Intel SGX. In *Network and Distributed System Security Symposium (NDSS).*

[22] Adil Ahmad, Juhee Kim, Jaebaek Seo, Insik Shin, Pedro Fonseca, and Byoungyoung Lee. 2021. CHANCEL: Efficient Multi-client Isolation Under Adversarial Programs. In *Network and Distributed System Security Symposium (NDSS).*

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

[24] Mohammed Alhamad, Tharam Dillon, and Elizabeth Chang. 2010. Conceptual SLA framework for cloud computing. In *4th IEEE International Conference on Digital Ecosystems and Technologies.*

[25] Amazon. [n. d.]. https://aws.amazon.com/lambda/pricing/.

[26] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[27] Adam Barth, Charles Reis, Collin Jackson, and Google Inc. [n. d.]. Google Chrome Team.

[28] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2014. Shielding Applications from an Untrusted Cloud with Haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[29] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2018. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *USENIX Security Symposium (USENIX Security).*

[30] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX Workshop on Offensive Technologies (WOOT).*

[31] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium (USENIX Security).*

[32] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A Library for Support Vector Machines. *ACM Transactions on Intelligent Systems and Technology* (2011).

[33] Stephen Checkoway and Hovav Shacham. 2013. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[34] Yuan Chen, Jiaqi Li, Guorui Xu, Yajin Zhou, Zhi Wang, Cong Wang, and Kui Ren. 2022. SGXLock: Towards Efficiently Establishing Mutual Distrust Between Host Application and Enclave for SGX. In *USENIX security symposium (USENIX Security).*

[35] Yueqiang Cheng, Zhi Zhang, and S. Nepal. 2018. Still Hammerable and Exploitable: on the Effectiveness of Software-only Physical Kernel Isolation. *ArXiv* (2018).

[36] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC).*

[37] Jonathan Corbet. 2009. Seccomp and sandboxing. *LWN. net, May* 25 (2009).

[38] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained.. In *IACR Cryptology ePrint Archive.*

[39] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium (USENIX Security).*

[40] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2015. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *Network and Distributed System Security Symposium (NDSS).*

[41] Rongzhen Cui, Lianying Zhao, and David Lie. 2022. Emilia: Catching Iago in Legacy Code. In *Network and Distributed System Security Symposium (NDSS).*

[42] Pubali Datta, Isaac Polinsky, Muhammad Adil Inam, Adam Bates, and William Enck. 2022. {ALASTOR}: Reconstructing the Provenance of Serverless Intrusions. In *31st USENIX Security Symposium (USENIX Security 22).*

[43] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. 2010. Privilege Escalation Attacks on Android. In *International Conference on Information Security (ICS).*

[44] Chuntao Dong, Qingni Shen, Xuhua Ding, Daoqing Yu, Wu Luo, Pengfei Wu, and Zhonghai Wu. 2022. T-Counter: Trustworthy and Efficient CPU Resource Measurement using SGX in the Cloud. *IEEE Transactions on Dependable and Secure Computing* (2022).

[45] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2021. Scalable Memory Protection in the PENGLAI Enclave. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI).*

[46] Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. 2004. Ostia: A Delegating Architecture for Secure System Call Interposition.. In *Network and Distributed System Security Symposium (NDSS).*

[47] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. 2019. AccTEE: A WebAssembly-Based Two-Way Sandbox for Trusted Resource Accounting. In *International Middleware Conference (Middleware).*

[48] Google. [n. d.]. https://cloud.google.com/functions/pricing.

[49] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium (USENIX Security).*

[50] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA).*

[51] Jinyu Gu, Bojun Zhu, Mingyu Li, Wentai Li, Yubin Xia, and Haibo Chen. 2022. A {Hardware-Software} Co-design for Efficient {Intra-Enclave} Isolation. In *31st USENIX Security Symposium (USENIX Security 22).*

[52] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism* (2005).

[53] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. 2013. InkTag: Secure Applications on an Untrusted Operating System. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[54] Ruirui Huang and G Edward Suh. 2010. Ivec: off-chip memory integrity protection for both security and reliability. *ACM SIGARCH Computer Architecture News* (2010).

[55] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. 2016. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[56] IBM. [n. d.]. https://cloud.ibm.com/functions/learn/pricing.

[57] Seongwook Jin, Jinho Seol, Jaehyuk Huh, and Seungryoul Maeng. 2015. Hardware-Assisted Secure Resource Accounting under a Vulnerable Hypervisor. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*.

[58] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. 2020. COIN Attacks: On Insecurity of Enclave Untrusted Interfaces in SGX. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[59] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. 2009. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating systems principles (SOSP)*.

[60] Youngjin Kwon, Alan M. Dunn, Michael Z. Lee, Owen S. Hofmann, Yuanzhong Xu, and Emmett Witchel. 2016. Sego: Pervasive Trusted Metadata for Efficiently Verified Untrusted System Services. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[61] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *European Conference on Computer Systems (EuroSys)*.

[62] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. 2017. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security Symposium (USENIX Security)*.

[63] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX security symposium (USENIX Security)*.

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

[65] Weijie Liu, Hongbo Chen, XiaoFeng Wang, Zhi Li, Danfeng Zhang, Wenhao Wang, and Haixu Tang. 2021. Understanding TEE Containers, Easy to Use? Hard to Trust. *arXiv preprint arXiv:2109.01923* (2021).

[66] Microsoft. [n. d.]. https://azure.microsoft.com/en-us/pricing/details/functions/.

[67] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. 2020. CopyCat: Controlled {Instruction-Level} Attacks on Enclaves. In *USENIX Security Symposium (USENIX Security)*.

[68] Joongun Park, Naegyeong Kang, Taehoon Kim, Youngjin Kwon, and Jaehyuk Huh. 2020. Nested Enclave: Supporting Fine-grained Hierarchical Isolation with SGX. In *International Symposium on Computer Architecture (ISCA)*.

[69] Weizhong Qiang, Zezhao Dong, and Hai Jin. 2018. Se-lambda: Securing privacy-sensitive serverless applications using sgx enclave. In *International Conference on Security and Privacy in Communication Systems*.

[70] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security Symposium (USENIX Security)*.

[71] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. 2011. DRAMSim2: A cycle accurate memory system simulator. *IEEE computer architecture letters* (2011).

[72] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer architecture news* (2013).

[73] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *USENIX Security Symposium (USENIX Security)*.

[74] Mark Seaborn and Thomas Dullien. 2015. Exploiting the DRAM rowhammer bug to gain kernel privileges. https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[75] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs.. In *Network and Distributed System Security Symposium (NDSS)*.

[76] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[77] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Network and Distributed System Security Symposium (NDSS)*.

[78] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. 2016. Preventing your faults from telling your secrets. In *ACM on Asia Conference on Computer and Communications Security (Asia CCS)*.

[79] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. PANOPLY: Low-TCB Linux Applications with SGX Enclaves. In *Network and Distributed System Security Symposium (NDSS)*.

[80] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W Fletcher. 2019. Microscope: Enabling microarchitectural replay attacks. In *International Symposium on Computer Architecture (ISCA)*.

[81] Jakub Szefer. 2019. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *Journal of Hardware and Systems Security* (2019).

[82] Meysam Taassori, Ali Shafiee, and Rajeev Balasubramonian. 2018. VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[83] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. 2018. Switchless Calls Made Practical in Intel SGX. In *Workshop on System Software for Trusted Execution (SysTEX)*.

[84] Bohdan Trach, Rasha Faqeh, Oleksii Oleksenko, Wojciech Ozga, Pramod Bhatotia, and Christof Fetzer. 2020. T-Lease: A Trusted Lease Primitive for Distributed Systems. In *ACM Symposium on Cloud Computing (SoCC)*.

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

[86] Chia-Che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX Annual Technical Conference (ATC)*.

[87] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *USENIX security symposium (USENIX Security)*.

[88] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security Symposium (USENIX Security)*.

[89] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. 2019. SGXJail: Defeating Enclave Malware via Confinement. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*.

[90] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. 2016. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *USENIX Security Symposium (USENIX Security)*.

[91] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy (S&P)*.

[92] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *IEEE Symposium on Security and Privacy (S&P)*.

[93] Jason Zhijingcheng Yu, Shweta Shinde, Trevor E Carlson, and Prateek Saxena. 2022. Elasticlave: An Efficient Memory Model for Enclaves. In *USENIX security symposium (USENIX Security)*.

[94] Xiaoyong Zhang, WU Ruizhen, Mingming Wang, and Lin Wang. 2019. A high-performance parallel computation hardware architecture in ASIC of SHA-256 hash. In *21st International Conference on Advanced Communication Technology (ICACT)*.

[95] Shixuan Zhao, Pinshen Xu, Guoxing Chen, Mengya Zhang, Yinqian Zhang, and Zhiqiang Lin. 2023. Reusable Enclaves for Confidential Serverless Computing. In *32nd USENIX Security Symposium (USENIX Security 23)*.