WAVEN: WebAssembly Memory Virtualization for Enclaves

Weili Wang*
Southern University of Science and Technology
12032870@mail.sustech.edu.cn

Honghan Ji
ByteDance Inc.
jihonghan.12138@bytedance.com

Peixuan He
ByteDance Inc.
hepeixuan.patrick@bytedance.com

Yao Zhang
ByteDance Inc.
zhangyao.crypto@bytedance.com

Ye Wu
ByteDance Inc.
wuye.2020@bytedance.com

Yinqian Zhang*†
Southern University of Science and Technology
yinqianz@acm.org

***Abstract*—The advancement of trusted execution environments (TEEs) has enabled the confidential computing paradigm and created new application scenarios for WebAssembly (Wasm). "Wasm+TEE" designs achieve in-enclave multi-tenancy with strong isolation, facilitating concurrent execution of untrusted code instances from multiple users. However, the linear memory model of Wasm lacks efficient cross-module data sharing and fine-grained memory access control, significantly restricting its applications in certain confidential computing scenarios where secure data sharing is essential (*e.g.*, confidential stateful FaaS and data marketplaces). In this paper, we propose WAVEN (WebAssembly Memory Virtualization for ENclaves), a novel WebAssembly memory virtualization scheme, to enable memory sharing among Wasm modules and page-level access control. We implement WAVEN atop WAMR, a popular Wasm runtime for TEEs, and empirically demonstrate its efficiency and effectiveness. To the best of our knowledge, our work represents the first approach that enables cross-module memory sharing with fine-grained memory access control in Wasm.**

I. INTRODUCTION

Trusted execution environments (TEEs) are processor extensions that provide applications with secure execution environments that protect the confidentiality and integrity of their private data and code, such that even malicious system software or administrators cannot access the memory regions protected by TEEs. This feature enables a new computing paradigm where sensitive computations can be performed on untrusted computing platforms, which is commonly called *confidential computing*.

The TEE technology has gained significant traction in recent years. Both commodity and open-source processors have TEE support. According to the abstraction they present to the users, TEEs can be categorized into user-space TEEs and VM-based TEEs. Prominent examples of user-space TEEs are Intel SGX [1] and Sanctum [2], which partitions the memory space of an application into trusted regions (called *enclaves*) and untrusted regions. Compared to VM-based TEEs, user-space TEEs offer a significantly smaller trusted computing base (TCB), reducing the attack surface. Particularly, Intel SGX is the earliest commodity user-space TEE that is available on the processor market and receives the widest adoption in both academia and industry.

Many confidential computing platforms have been built atop SGX, such as confidential Function-as-a-Service (FaaS) [3], [4], [5] and privacy-preserving data analysis [6], [7]. Hosting multiple mutually distrustful workloads inside a single user-space enclave is desired for better efficiency as inter-enclave scheduling incurs high overhead [8], [5]. For instance, Zhao *et al.* [5] report that reusing an enclave is averagely 159× faster than creating a new enclave, and Ahmad *et al.* [8] show that hosting multiple clients in a single enclave performs 4.06-53.70× better than hosting each client in a separate enclave. This practice demands in-enclave multi-tenancy with strong security isolation, to prevent a malicious tenant from harming the others sharing the same enclave.

However, there is no established support for in-enclave multi-tenancy in SGX, although several efforts have been made to address this issue. One line of research [9], [10], [11], [12] manages to utilize hardware primitives (*e.g.*, Intel MPX [13] and MPK [14]) to achieve intra-enclave isolation. Depending on additional specialized hardware support other than SGX renders these solutions less practical. Another line of research explores the use of software fault isolation (SFI) [6], [15], [7], [8] in enclaves. Among these works, the most prevalent direction is to integrate WebAssembly (Wasm) [16] with SGX enclaves, due to its formally specified and verified nature [17], [18], as well as its well-established ecosystem supported by industry leaders [19].

Wasm is a novel portable and efficient binary format. Wasm runtimes provide SFI-based sandboxes for untrusted code instances (*i.e.*, Wasm modules), ensuring strong isolation between them. Integrating Wasm runtimes with SGX enclaves holds great promise for confidential computing platforms that serve multiple mutually distrustful users. Notably, there have

*Affiliated with the Research Institute of Trustworthy Autonomous Systems and the Department of Computer Science and Engineering
†Corresponding Author

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.230746
www.ndss-symposium.org

been numerous academic prototypes [4], [20], [21], [15], [22], [5] and industry products [23] that explore this direction. The "Wasm+SGX" combination can be traced back to the proposal of Ryoan [6], which uses Native Client [24], the predecessor of Wasm, to isolate different workloads within an enclave. Similarly, systems like TWINE [15] and AccTEE [20] establish in-enclave sandboxes using Wasm runtimes. Based on in-enclave Wasm runtimes, confidential computing platforms such as Se-Lambda [4], S-FaaS [21], Teaclave [3] have been developed to provide confidential FaaS services.

Despite the widespread adoption of in-enclave Wasm, its memory management is insufficient in confidential computing settings. The root cause is the over-simplified linear memory model of Wasm, which treats the entire memory as a single indivisible byte array and does not involve any memory sharing or protection mechanism. While this design choice may have been reasonable for Wasm's original purpose of lightweight Web applications, it proves unsuitable for standalone environments, particularly within enclaves.

Such insufficiency is particularly noteworthy in the following aspects: First, Wasm does not support cross-module memory sharing—it only allows a module to export its entire memory instead of specific data regions. This is not flexible enough for practical use cases. Second, a Wasm module's entire linear memory is writable, lacking read-only regions. This coarse-grained memory access control not only prevents secure data sharing among modules (shared memory can be written by all) but also exposes security vulnerabilities [25]. These deficiencies greatly restrict in-enclave Wasm. To ensure confidentiality in platforms (*e.g.*, stateful FaaS [26] and data marketplaces [27]) where multiple users concurrently access the shared data with different permissions, a memory management scheme with efficient memory sharing and fine-grained access control is needed.

The Wasm community is aware of the limitations of the linear memory model and plans to address them. The official documentation explicitly states that *"In the future, support for multiple linear memory sections and finer-grained memory operations (e.g., shared memory, page protection, large pages, etc.) will be implemented"* [28]. However, to date, there is no proposal for the memory protection feature in the WebAssembly proposal list [29], and the multi-memory proposal [30] related to shared memory is not fully developed and lacks compiler support [31], [32]. While a non-official shared memory scheme has been implemented in Faasm [33], it relies on a system call that is impossible to implement in enclaves. As such, the lack of efficient memory sharing and fine-grained memory access control in Wasm remains a significant challenge for enclaves.

In this paper, we seek to design a memory virtualization scheme for in-enclave Wasm runtimes, aiming to achieve efficient memory sharing across multiple Wasm modules and fine-grained memory access control. However, fulfilling these goals poses several challenges.

- *Complexity:* Memory virtualization that supports in-enclave multi-tenancy, especially scenarios requiring efficient memory sharing and fine-grained access control, demands complicated system design.
- *Efficiency:* Memory virtualization in SGX without hardware and OS support requires the implementation of a software memory management unit (MMU) in the Wasm runtime, which introduces significant performance overhead.
- *Compatibility:* The memory virtualization scheme must be compatible with Wasm specification, supporting the execution of traditional Wasm modules without modification.

In response to these challenges, we propose WAVEN, **W**eb**A**ssembly Memory **V**irtualization for **EN**claves, a Wasm memory virtualization scheme that implements a software MMU for paging. *Firstly*, WAVEN implements a single-level page table and a software MMU logic for page table walks. Each memory access incurs only one page table visit, minimizing the performance overhead due to page table walks. *Secondly*, WAVEN leverages novel schemes like exception pages and memory paddings to reduce the time-consuming boundary checks for each memory access, optimizing the performance of WAVEN. *Thirdly*, to support fine-grained memory access control, WAVEN leverages two separate page tables for read and write operations, enforcing read-only memory semantics while avoiding heavy-weight permission checks. *Finally*, WAVEN remains compatible with the Wasm specification, allowing unmodified Wasm modules to run.

We implement WAVEN atop WAMR [34] and evaluate its performance inside SGX, the most widely deployed user-space TEEs. The evaluation results show that memory virtualization introduces moderate overhead—the geometric mean of overheads in PolyBench [35] is only 10.42%. In the STREAM benchmark performing memory stress test, WAVEN even outperforms the vanilla WAMR implementation in three of the four test cases. In two typical confidential computing workloads, namely confidential database and privacy-preserving machine learning, WAVEN incurs 12.52% and 6.14% overheads, respectively, indicating its suitability in common confidential computing scenarios. For further demonstration of the applicability of WAVEN, we also develop a confidential computing platform prototype atop WAVEN and compare it to the one without memory virtualization. The results confirm that our scheme can significantly improve performance (up to 2.4× faster in high concurrency settings) in different memory sharing scenarios, including multi-write multi-read and multi-read sharing patterns.

In sum, the paper makes the following contributions:

- The first practical design of Wasm memory virtualization for in-enclave multi-tenancy, enabling efficient page-level cross-module memory sharing and access control.
- Multiple performance optimization schemes in WAVEN, such as exception pages and memory paddings, exemplifying a software MMU design with minimal performance overhead.
- A prototype implementation and systematic evaluation of WAVEN, as well as evaluation on data sharing scenarios, demonstrating its practical applicability.

II. Background

A. WebAssembly in A Nutshell

WebAssembly is a binary instruction format and a portable compilation target for high-level languages, functioning as a fast, sandboxed stack machine. A Wasm program, also known as a module, consists of a linear memory, variables, functions, and tables. A module will get executed when its exported functions are invoked, and the runtime is responsible for managing its operand stack.

1) Software Fault Isolation: The design of Wasm incorporates SFI techniques to achieve control flow integrity and memory isolation among modules.

Control flow integrity (CFI). Unlike traditional CFI enforcements that involve time-consuming runtime instrumentation, Wasm implements CFI implicitly through structured control flow. This prevents modules from jumping to arbitrary targets. A function table, generated during load time and managed by the runtime, indexes all Wasm functions. It is utilized to resolve direct and indirect function calls. Consequently, a Wasm module can only call functions that are known to the runtime. Additionally, there are function type checks for indirect calls to ensure that the function being called indirectly possesses the correct type signature as recorded in the function table. Lastly, the operand stack consisting of call frames is managed by the runtime, and direct access to the operand stack by modules is not allowed, which protects return addresses.

Memory isolation. Memory isolation ensures that each Wasm module can only access its linear memory, which is a contiguous byte array allocated by the runtime. Within the Wasm module, memory addresses are expressed as 32-bit values called Wasm addresses, ranging from `0x00000000` to `0xffffffff`. These addresses are then translated to 64-bit virtual addresses within the runtime process.

When allocating memory for a module, the runtime records the base address of its linear memory. When the module accesses a Wasm address, it is translated to an effective virtual address by adding the base address and the Wasm address. The runtime then checks the legality of the effective virtual address to ensure it falls within the memory range of the module. This type of boundary checking can be implemented using either hardware traps or software comparisons. Although software boundary checks are easier to implement, they introduce more overhead as they require an additional comparison for every memory access. In the hardware-based approach, the runtime marks memory outside the module's memory range as inaccessible, and hardware raises an exception when there is illegal access. This approach eliminates the need for explicit boundary checks and thus is more efficient, but it necessitates hardware support, which may not be available on all platforms (*e.g.*, not supported in SGX enclaves).

Wasm system interface (WASI). WASI is a modular system interface designed to be platform-independent. It acts as a bridge between Wasm modules and the host operating system, enabling unprivileged modules to access OS functionalities like file systems and networking. WASI specifies a set of standardized interfaces for modules to use system utilities, and Wasm runtimes are required to implement the interfaces based on the host environment. WASI improves portability and reduces security risks by limiting the attack surface.

2) Wasm Execution Modes: Most Wasm runtimes, such as WAMR [34] and Wasmtime [36], support three execution modes for Wasm modules:

Interpretation. The Wasm module is executed by a Wasm interpreter within the runtime. The interpreter reads Wasm bytecode and executes instructions one after another. Due to the interpretation overhead, this mode is the slowest.

Just-in-Time (JIT) compilation. The Wasm runtime loads and compiles bytecode into native code for execution. JIT compilation is dynamically performed, but the overhead is offset by fast native code execution, leading to improved performance. Different runtimes employ various JIT compilation schemes, like tiered compilation in WAMR and speculative compilation in Wasmtime, to enhance performance.

Ahead-of-Time (AOT) compilation. The AOT mode is the fastest among the three modes. In this mode, the AOT compilers compile the Wasm bytecode into native code in advance, and the Wasm runtime directly executes the compiled code. Different Wasm runtimes specify different AOT formats and have their own AOT compilers.

B. User-space TEEs

User-space and VM-based TEEs are two types of TEEs. User-space TEEs (*e.g.*, Intel SGX [1]) allow unprivileged user applications to create isolated memory regions, dubbed *enclaves*, in their own address spaces, and load sensitive code and data into these enclaves such that their execution is protected by the TEE hardware. Examples of VM-based TEEs include Intel TDX [37], AMD SEV [38] and ARM CCA [39]. They provide TEE users with a virtual machine abstraction, allowing unmodified applications to run inside of the so-called confidential VMs directly.

Intel Software Guard eXtension. Intel SGX provides hardware-based security features to protect sensitive data and code execution. It allows developers to create isolated enclaves, encrypted memory regions that are immune to attacks from other processes, the operating system, and even partial physical intrusion. SGX also supports local and remote attestation, enabling enclaves to verify counterparts on the same platform and allowing remote parties to verify an enclave's identity and memory integrity.

SGX software development entails a tedious process of dividing applications into the trusted part and the untrusted part, and an enclave can only run a single application. To ease the development and support multi-tenancy in an enclave, embedding a runtime in the enclave is a common practice, with Wasm being a popular choice. Examples include TWINE [15], [22] and AccTEE [20], where applications are compiled into Wasm modules and then executed in sandboxes in the enclave.

User-space TEEs are here to stay. The deprecation of Intel SGX on desktop CPUs [40] and the growing popularity of VM-based TEEs, such as AMD SEV [41], [42], seem to have indicated that VM-based TEEs will be the dominating form of TEEs in the future. However, this is not the case.

Firstly, Intel did not abandon SGX. In fact, Intel continues to view SGX as an indispensable element in the realm of confidential computing and will continue to provide SGX support on server-grade processors [43]. Even on TDX machines, SGX enclaves are still used to establish the chain of trust and support remote attestation for TDX [44]. The deprecation of desktop SGX is in part due to its mismatch with confidential computing scenarios. Moreover, cloud providers have also heavily invested in SGX technology. For instance, Microsoft Azure and Alibaba Cloud both provide SGX-enabled VMs [45], [46], and IBM Cloud offers bare metal servers with SGX support [47].

Secondly, user-space TEEs have smaller TCB and hence are better suited for scenarios with high security demand. Particularly, as a VM-based TEE must incorporate an entire operating system into its TCB, confidential applications relying on VM-based TEEs tend to have a significantly larger attack surface. As a prominent example, Intel has to make a great effort to tailor a Linux kernel for TDX, in order to mitigate some known attack vectors [48]. This drawback has been recognized and efforts have been made to reduce the TCB of VM-based TEEs by providing user-space enclaves inside a confidential VM [49], [50], [51], [52].

III. Motivation

We first present the system model and example use cases to highlight the importance of efficient data sharing and fine-grained memory access control in an in-enclave Wasm runtime. We then discuss the limitations of the current Wasm linear memory model that hinder its adoption in enclaves, and finally introduce the inspiration behind WAVEN.

A. System Model

Fig. 1 shows the system model that motivates the design of WAVEN. Specifically, we consider a Wasm-based confidential computing platform running in an SGX enclave, and there are three mutually distrustful roles: the platform owner, data providers, and data consumers. Data providers and consumers are users who interact with the platform to share data and perform computation. For a piece of data, the data provider and the data consumer can be the same or different entities, depending on the scenario.

In the platform, multiple tasks from the same or different data consumers can concurrently access the same data, but only the data provider or authorized consumers can modify the data. This computing paradigm mandates efficient data sharing with fine-grained access control across Wasm modules. The platform's security goals include ensuring (1) that the data providers' data cannot be leaked or used without their consent, and (2) that the confidentiality and integrity of the execution of the data consumers' tasks are preserved.

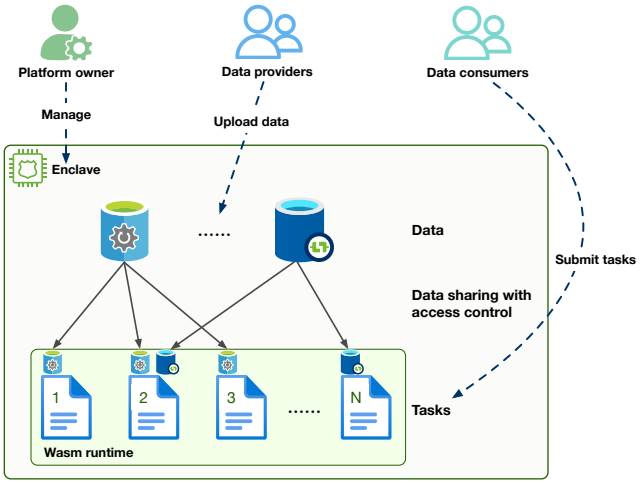


Figure 1: Illustration of the system model. The platform owner manages the enclave and the platform, data providers upload data, and data consumers submit computation tasks.

Platform owner. The platform owner develops the platform program and manages the enclave hosting the platform. The platform provides services to data providers and data consumers, such as data storage, computation execution, and access control enforcement. The platform utilized a Wasm runtime to enable mutually distrustful data consumers to execute their computation tasks in in-enclave Wasm sandboxes. The platform owner will make the source code of the platform program available to the users for auditing. As such, the users can verify the correctness of the platform program and the enclave through remote attestation although they do not trust the owner. Note that only the platform program logic is public but in-enclave data and tasks remain confidential.

Data providers. Data providers are users who own private data and want to share it with others or to compute on it themselves using the platform's computing power. Data providers upload data to the platform and specify the access control policies.

Data consumers. Data consumers submit their computation tasks to the platform in the form of Wasm modules and obtain the computation results. A task can be a machine learning model or a data processing script that operates on the data stored in the platform. The platform validates whether the user is authorized to access the data before executing the task.

B. Threat Model

We assume that the underlying SGX enclave fulfills its security guarantees, namely, the confidentiality and integrity of in-enclave data and code are well protected. The platform owner, data providers, and data consumers, however, can be malicious and collude with one another to violate the platform's security goals. Specifically, we consider the following threats:

Insecure platform program. The platform owner may develop a platform program with backdoors that steals or tampers with users' data or code. As the source code of the platform program is public, the users can audit the platform program to detect such misbehavior.

Arbitrary input data. Data providers can upload arbitrary data to the platform. The platform makes no assumption about the data's correctness or usefulness. Data providers can also launch Denial-of-service (DoS) attacks by uploading a large amount of data, but the impact is minor as the platform usually charges the data providers for the storage space in practice.

Arbitrary input modules. The data consumers can generate Wasm modules with arbitrary computation logic and submit them to the platform. DoS attacks can be closed by charging the data consumers for the computation resources.

The platform owner controlling the system software may refuse to provide services to users or launch side-channel attacks [53] to compromise the enclave's security. We do not consider such threats.

C. Example Use Cases

The considered system model plays a significant role in various confidential computing scenarios. We present two example target scenarios to illustrate the importance of data sharing and fine-grained memory access control in an in-enclave Wasm runtime.

1) Confidential Stateful FaaS: FaaS (serverless) computing paradigm is predicted to dominate the future of cloud computing [54], and mainstream cloud providers have already launched their FaaS platforms (*e.g.*, AWS Lambda [55] and Google Cloud Functions [56]). FaaS supports stateless and stateful applications. Recently, stateful FaaS has been gaining traction due to its ability to maintain state across function invocations, enabling more complex applications [26], [33], [57], [58].

In this setting, FaaS users may compute on their data or public data. When processing private data, the user is both a data provider and a data consumer—it can upload the data and the computation task to the platform and then leverage the auto-scaling feature of FaaS to execute the task more efficiently. When processing public data like public datasets organized by the FaaS platform (*e.g.*, AWS Lambda users can use public datasets maintained in AWS Open Data [59]), the user is only a data consumer.

Need for data sharing. Either private or public data, the to-be-computed data should be shared among multiple function instances. A FaaS user can create a large number of function instances to process a dataset in parallel to speed up the computation, with each function instance accessing the partial or entire data. Data sharing across function instances enables direct computation on the original data without duplication and can significantly enhance the performance, as demonstrated in prior works [33], [26].

Need for access control over shared data. For private data, the user owns the data but may want to keep the data unchanged in computation. For example, unintended bugs in function instances may corrupt the data and deviate the computation. With access control, the user can specify that the data is read-only for the function instances, preventing accidental data modification, similar to the use of constants in programming languages. For public data shared among different users, access control becomes more critical—a user can affect other users' computation by tampering with the shared data if there is no access control. To sum up, the platform needs to enforce fine-grained memory access control over the shared data to prevent unauthorized data modification.

Status quo. There have been several Wasm-based FaaS platforms [20], [5] running in enclaves, showing the popularity of this combination. However, these platforms lack efficient data sharing mechanisms and fine-grained memory access control, which are important for stateful FaaS applications, as discussed above. The system model we considered can address these issues and fill the gap in the current in-enclave Wasm-based FaaS platforms.

2) Secure Data Marketplaces: Data marketplaces facilitate data exchange and utilization and thus are of great value in this big data era. The global data marketplace platform market is expected to grow from $1,192.1 million in 2023 to $7,312.5 million by 2032, with a growth rate of 22.33% during 2024-2032 [60]. Notable industry data marketplaces include Snowflake Marketplace [61] and AWS Data Exchange [62].

To build secure data marketplaces that protect data privacy and integrity, numerous designs(*e.g.*, [63], [64], [65]) combining blockchain technology and SGX have been proposed. In these designs, the platform owner acts as the marketplace operator, who manages the enclave hosting the marketplace and provides services to data sellers and data buyers. The data sellers and buyers are the data providers and consumers in our system model, respectively.

Need for data sharing with access control. In data marketplaces, sellers upload their data, and buyers can purchase the data in the blockchain and compute on it. The computation is either on the raw data [64] or on the statistical features extracted from the raw data [63]. In both cases, the computation is usually performed by the blockchain execution engine (*e.g.*, EVM [66] and Wasm). In the marketplace, it is common for a piece of data to be sold to multiple buyers, necessitating data sharing across buyers' computation instances to enhance performance. Otherwise, each buyer must copy the purchased data before computation, which not only wastes memory but also slows down the process. Moreover, fine-grained memory access control over the shared data is essential—the seller should have the privilege to modify the data while buyers should only be able to read the shared data.

Status quo. The aforementioned confidential data marketplace designs [63], [64], [65] do not support data sharing with fine-grained access control inside the enclave, which is addressed in our system model. The increasing adoption of Wasm as the execution engine in blockchains, as evidenced by its use in several blockchains [67], [68], [69], further validates the practicality of our system model for this case.

D. Limitations of Wasm Linear Memory

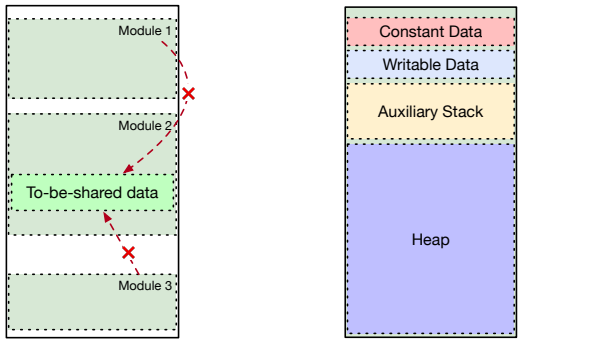
Wasm's linear memory, a contiguous block of memory that is allocated and managed by the runtime environment,

is a crucial component of the Wasm virtual machine that allows programs to access and store data. Although linear memory has been a *de facto* standard that has been adopted by most WebAssembly runtimes, it has several limitations when deployed in enclaves.

Poor data sharing. A Wasm module has at most one linear memory and it can export its memory to other modules or import another module's memory. This enables memory sharing among different modules, but it is rather inflexible and impractical—what developers need to share is usually a specific data region instead of the entire linear memory. Fig. 2a illustrates a scenario in which a module intends to share its data with two other modules. However, with the current linear memory model, these two modules are unable to directly access the shared data; explicit memory copies are necessary. The draft multi-memory proposal [30] allows a module to have multiple memory regions, aiming to facilitate data exchange among modules. However, this approach has its own limitations. First, the granularity with which memory regions can be shared is very coarse. It is not possible to share a fraction of a memory region. This becomes a more severe issue when a module needs to share different data with different modules. Second, compiling programs written in high-level languages into Wasm bytecode with multiple memory support is challenging, as these languages typically assume a single memory model. This issue remains unresolved since the introduction of the multi-memory proposal in 2019 [70]. To date, developers are still unable to compile their code written in other languages to Wasm modules with multiple memories due to the lack of compiler support [31], [32].

Some Wasm-based FaaS frameworks implement their own data sharing mechanisms. For example, FAASM [33] leverages the Linux `mremep` system call to make a module's partial linear memory point to the shared data. However, this approach is not applicable to user-space TEEs like SGX. This is because SGX enclaves rely on the untrusted OS to conduct memory allocation and page table management. To ensure security, SGX hardware maintains Enclave Page Cache Map (EPCM) inside the protected memory, recording the allocation information (*e.g.*, the owner enclave, access permission) of EPC pages. An EPCM entry contains the virtual address used to access the corresponding EPC page, which ensures that an EPC page can only be accessed by a unique virtual address[71, Sec. 5.2.3]. As such, sharing memory via `mremap` will map different virtual addresses to the same EPC page of the enclave memory, leading to unsuccessful EPCM checks and thus enclave termination.

Coarse-grained memory access control. Wasm restricts each module to access its own memory region, but it does not provide fine-grained memory access control, such as read-only memory. As shown in Fig. 2b, the data region, auxiliary stack, and heap are all writable. The entirely writable memory poses significant challenges to secure data sharing across modules within the enclave. In this design, the module sharing its data with other modules lacks control over the access permissions



(a) Inefficient data sharing. (b) Entirely writable linear memory.

Figure 2: Limitations of the linear memory model.

of the shared data, which potentially leads to unauthorized data modification. Furthermore, the coarse-grained memory access control may result in undefined behavior or even security vulnerabilities. Lehmann *et al.* [25] have shown an end-to-end attack targeting standalone Wasm runtimes. Due to the lack of read-only memory, the attacker can tamper with the victim module's constant data by contriving malicious inputs, and successfully write arbitrary files in the host file system via WASI. This case suggests that entirely writable memory is not secure, even in regular environments, let alone in enclaves where the security requirements are even more stringent.

E. Solution: WebAssembly Memory Virtualization

Modules hosted inside a Wasm runtime are similar to processes running in an OS, and alike an OS kernel, the runtime needs to allocate memory for modules, and manage their execution. The evolvement of memory management in OSs, especially the transition from direct allocation on physical memory (similar to the linear memory model of Wasm) to memory paging, provides valuable insights for better memory management in Wasm runtimes. We draw inspiration from OSs' experience and propose a memory virtualization scheme for Wasm runtimes, which is based on memory paging.

Paging in OS functions to map the virtual addresses of processes to physical addresses in memory. In the case of Wasm modules, which operate within the runtime process, we refer to Wasm addresses as "virtual addresses" and the runtime's virtual addresses as "physical addresses". Specifically, a module's entire Wasm address space and the runtime's partial virtual address space are divided into fixed-size pages. Each Wasm page is assigned a unique page index, which is then mapped to a virtual page. To record mappings, we leverage page tables, which serve as lookup tables for address translation. The runtime is responsible for creating and managing page tables, and each Wasm module has its own page table.

IV. WAVEN DESIGN

A. Overview

WAVEN is a **W**eb**A**ssembly Memory **V**irtualization scheme for **EN**claves that aims to support multiple WebAssembly

modules to efficiently share their memories with fine-grained access control. WAVEN has the following design goals:

- **Practicality.** While enabling flexible memory management (*i.e.*, memory sharing and access control), the scheme should also provide the same linear memory abstraction to Wasm modules and comply with the Wasm specification (as much as possible).
- **Security.** The scheme should ensure memory isolation as the linear memory does, that is, each Wasm module only accesses its own "virtual" linear memory.
- **Performance.** The introduced performance degradation should be minimal.

To meet the design goals with a paging-based memory virtualization scheme, however, we still need to address the following challenges:

- **C1: Complexity.** Designing a scheme supporting in-enclave multi-tenancy with efficient memory sharing and fine-grained access control could be complex. Controlling the complexity of the scheme is key to ensure its practicality.
- **C2: Efficiency.** Deploying the scheme inside SGX enclaves without hardware and OS support mandates a software MMU implementation in the WebAssembly runtime. This could introduce significant overhead without proper optimizations.
- **C3: Compatibility.** The linear memory model is a crucial component of the WebAssembly specification, and introducing paging as an alternative while ensuring compatibility with Wasm poses challenges.

B. WebAssembly Paging

At the core of WAVEN is a novel paging scheme and an efficient software MMU design for Wasm.

Page size. Theoretically, WAVEN can support arbitrary page sizes, but we set the page size to 64KB in accordance with the WebAssembly specification. The specification specifies that the size of a Wasm module's memory should be a multiple of 64KB for portability. For 32-bit Wasm addresses, the higher 16 bits indicate the page index, while the lower 16 bits represent the offset within a page ($2^{16}\text{B} = 64\text{KB}$).

Page table. A Wasm module is limited to a maximum of 4GB of memory space, equivalent to 65536 memory pages. Page tables translate the Wasm module's 32-bit addresses (ranging from `0x00000000` to `0xFFFFFFFF`) to the runtime's 64-bit virtual addresses. WAVEN adopts single-level page tables. This choice reduces the performance overhead by performing only one page table walk per memory access. It also simplifies the design of memory sharing and access control. Therefore, the choice of single-level page tables helps to address challenge **C1** and **C2**. In this configuration, a page table consists of 65536 entries, each containing a 64-bit virtual address. Consequently, a single page table occupies a memory space of 512KB, which is small and acceptable.

Address translation. Address translation occurs when Wasm modules access memory addresses using related instructions(*e.g.*, `i32.load` and `i32.store`). To translate a 32-bit Wasm address, the Wasm runtime first extracts the higher 16 bits as the page index, looks up the page table to find the virtual address of this Wasm page, and finally calculates the virtual address to be accessed by adding the lower 16 bits (page offset) to the previously retrieved virtual address. Fig. 3 illustrates the paging scheme. The arrows represent page table entries that record the mapping from a module's Wasm pages to the runtime's virtual pages. Note that adjacent Wasm pages can be mapped to non-adjacent virtual pages. Paging introduces an extra memory read in the address translation process compared to the linear memory model, where the virtual address is calculated by simply adding the base address and the Wasm address.

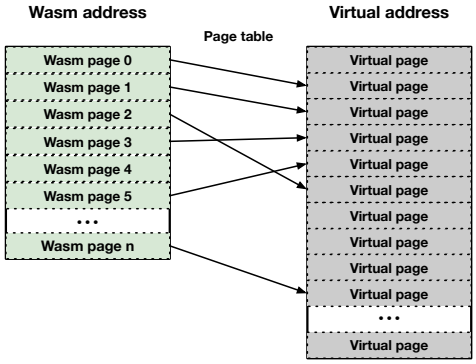


Figure 3: The paging scheme. Arrows denote the page table.

C. Memory Isolation

The linear memory model ensures memory isolation among Wasm modules using boundary checks. The runtime first checks whether the target Wasm address is accessible before translating it to a virtual address. Given that paging already incurs one additional memory read (page table lookup) in every address translation, adopting inefficient boundary checks (SGX only supports software boundary checks with high overhead) would further degrade the performance. To overcome this challenge (related to **C2**), we propose the use of *exception page* and *page padding* to optimize address translation and improve performance (as shown in Fig. 4).

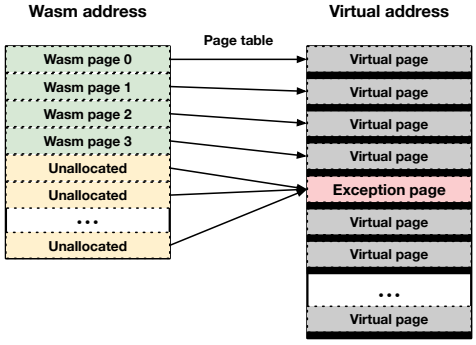


Figure 4: Illustration of the exception page (highlighted in red) and page padding (represented using black rectangles).

Exception page. In paging, memory isolation is enforced by manipulating page table entries, which control the translation

from Wasm addresses to the runtime's virtual addresses. During module instantiation, the runtime not only allocates initial memory pages but also creates a 64KB empty exception page for the module. For Wasm addresses within the initial memory space (legal accesses), the runtime sets the corresponding page table entries to the virtual addresses of the allocated memory pages. On the other hand, for Wasm addresses outside the initial memory space (out-of-bounds accesses), the runtime modifies the related entries to point to the exception page. As a result, when a module performs an out-of-bounds access, the address translation will direct it to its own exception page. This effectively prevents the module from interfering with the memory of another module or the runtime without explicit boundary checks.

Page padding. Similar to paging in modern computer architectures, WAVEN also needs to deal with unaligned memory accesses. Common unaligned accesses that entirely lie inside a 64KB page bring no trouble, while those crossing page boundaries need special handling. For instance, a `i32.load` instruction reading four bytes may start from the last byte of a Wasm page and end at the first three bytes of the next page. The high overhead of these cross-page accesses is twofold in WAVEN. First, it needs to check whether the memory access crosses two pages. Second, it needs to conduct address translation two times for unaligned accesses, resulting in two page table lookups.

By contrast, the probability of cross-page memory accesses is rather low. Compilers (*e.g.*, Emscripten [72] and WASI SDK [73]) emitting Wasm bytecode will align memory accesses automatically (aligned memory accesses never span two pages). In addition, not all unaligned memory accesses are cross-page accesses. Our compilation of the PolyBench [35] test suite further supports this observation—there are no cross-page accesses in all memory accesses of the 30 compiled Wasm modules.

Considering the prohibitive overhead and small probability of cross-page accesses, we discard support for them in WAVEN—we pad each page with several bytes to ensure that even cross-page accesses lie inside a single page. In this way, the runtime can always translate a Wasm address to a virtual address with only one page table lookup. Even in cross-page accesses, the runtime will not generate illegal memory accesses. Instead, it will access the padding bytes, effectively ensuring memory isolation. As one Wasm memory access instruction at most reads/writes eight bytes (we do not consider SIMD instructions at this time), the minimum padding size is seven. The empirical study reported in Sec. VI-B suggests that the 7B padding is a good choice.

With page padding, the correctness of execution without cross-page accesses is guaranteed whereas the one with cross-page accesses is not. This is reasonable as the compiled Wasm bytecode rarely accesses memory across page boundaries, and developers can still manually align memory accesses to avoid cross-page accesses with the help of WAVEN's debugging mode: To assist developers in identifying subtle bugs that may be caused by cross-page accesses, WAVEN offers a debugging mode to detect cross-page accesses. In the debugging mode, the runtime inserts one additional boundary check before each memory access, and informs the developer when a cross-page access is detected. As such, developers can locate the problematic code and manually align memory accesses, which is a common practice in architectures that do not mandate support for unaligned memory accesses, such as RISC-V [74] and ARM [75].

D. Memory Access Control

Wasm modules have their own page tables, allowing us to utilize isolated page tables to confine memory accesses of different modules. Building on this idea, we propose a memory access control solution using dual page tables, which does not involve time-consuming software permission checks and contributes to the address of challenge **C1** and **C2**.

Our approach involves using two separate page tables for read and write operations. The write page table handles memory writes and stores the virtual addresses of writable pages. On the other hand, the read page table is used for memory reads and stores the virtual addresses of readable pages. The page table entries for writable pages are the same in both page tables, while the entries for read-only pages differ.

In the read page table, the entries corresponding to read-only pages point to effective virtual addresses. In contrast, the entries in the write page table for read-only pages point to the address of an exception page. This setup allows the runtime to dynamically change page permissions by modifying the page table entries in response to requests from Wasm modules.

To illustrate this concept, consider the example of read-only pages shown in Fig. 5. In the module's read page table, Wasm page 3 is mapped to a normal virtual page. However, in the write page table, it is mapped to the exception page. As a result, the module can read the content of Wasm page 3 but cannot modify it.

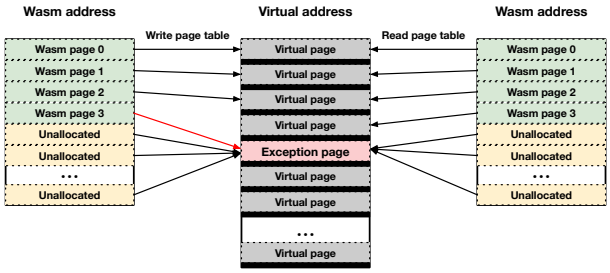


Figure 5: An example of read-only pages. Page 3 is read-only and mapped to the exception page in the write page table.

This fine-grained access control is particularly useful in shared memory scenarios, where multiple Wasm modules may need to read shared content but should not be able to modify it without permission. Additionally, the introduction of read-only pages also enhances security by preventing Wasm modules from modifying constant (read-only) regions. In a previous study by Lehmann *et al.* [25], it is demonstrated that attackers

can write arbitrary files by exploiting buffer overflow attacks to modify constant data.

E. Memory Remapping

Unlike the `mremap` system call in Linux, which requires a transition to kernel space, memory remapping in WAVEN is simpler and can be done in user space at runtime. In WAVEN, a page table entry maps a Wasm page to a virtual page, allowing the runtime to easily remap a Wasm page to another virtual page by overwriting the page table entries. This memory remapping feature greatly facilitates shared memory among multiple WebAssembly modules, as modules can share data by mapping their Wasm pages to the same virtual pages.

Furthermore, by combining memory remapping with read-only pages, WAVEN can enforce read-only permissions on the shared memory region. This means that some modules can only read from the shared memory while others can modify it. This level of control is not achievable with shared memory implementations based on `mremap`, as there is no way to set read-only permissions for specific Wasm modules within a runtime process without specialized hardware support such as Intel MPK [14].

F. Summary

To overcome the challenges mentioned in Sec. IV-A, we propose WAVEN, a novel paging scheme with an efficient software MMU design. The use of single-level page tables and dual-page-table-based memory access control helps to tackle challenges **C1** and **C2**. To further address challenge **C2**, we introduce the exception page and page padding. Furthermore, we address challenge **C3** by designing a scheme that only requires modifications to Wasm runtimes, and not to the Wasm applications themselves. This means that developers can still use their preferred compilers to compile high-level language code into Wasm bytecode without making any modifications.

V. IMPLEMENTATION

We implement WAVEN on top of WAMR [76] (commit `92c4bbe`) and SGX SDK 2.19, adding approximately 3,000 lines of code. WAVEN does not require changes to the Wasm bytecode itself. It uses existing LLVM toolchains, such as WASI SDK [73] and Emscripten [72], to generate the bytecode without any modifications.

Since WAMR supports Intel SGX out of the box, there is no need to port the implementation to SGX enclaves. Our prototype implementation primarily targets the fastest execution mode—AOT compilation. The AOT compiler in WAMR leverages LLVM to compile Wasm bytecode to AOT (native) code, and thus we modify the LLVM backend (version 15.0.7) to enable memory virtualization. WAVEN can be easily extended to support JIT compilation since both the AOT and JIT compilers share most of the codebase in WAMR. Additionally, supporting interpretation mode is straightforward as it only requires modification of the runtime, which is finished already in our prototype.

We also implement runtime interfaces for shared memory management, enabling the Wasm module developers (*i.e.*, the data consumers in the system model described in Sec. III-A) to use shared memory. Similar to WASI, these interfaces are implemented as runtime functions that can be invoked by Wasm modules, and they are exposed to developers in the form of function declarations. In development, the developers can include the header files that contain these function declarations to use these interfaces. The finished source code is first compiled to Wasm bytecode using the aforementioned existing tools, and then passed to our modified AOT compiler for generating the AOT code with memory virtualization support. Finally, the AOT code is executed in our modified WAMR runtime that supports memory virtualization. The above process is also illustrated in Fig. 6.

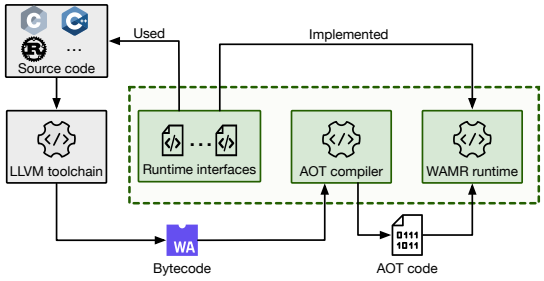


Figure 6: Illustration of the workflow for developers to generate and execute a Wasm module with WAVEN. Our implementation is enclosed in a dotted box.

We then detail the compiler modification for page-based address translation, runtime modification for page table management, and runtime interface implementation.

A. AOT Compiler Modification

The AOT compiler takes the bytecode of a Wasm module as input and produces native code as output. It first parses the bytecode and constructs the LLVM intermediate representation (IR) of the Wasm module. Then, the compiler performs optimizations and generates a more efficient IR, from which the native code is generated.

Address translation from Wasm addresses to the runtime's virtual addresses occurs during the parsing step. The AOT compiler parses the Wasm bytecode function by function. As the compiler cannot infer the runtime's virtual address space throughout the AOT compilation, it adds an extra argument to store running information into each Wasm function's parameter list. This facilitates address translation. For example, a Wasm function declared as `void func (arg0, arg1, ..., argN)` in the bytecode is parsed into an LLVM function with the declaration `void func (exec_env, arg0, arg1, ..., argN)` in the IR. The `exec_env` variable is a C language structure created by the runtime, which stores a module's running information, including the virtual address of the linear memory. When encountering WebAssembly memory access instructions (*e.g.*, `i32.load` and `i32.store`) within a function, the compiler inserts

address translation instructions (IR instructions). These instructions take a Wasm address as input, check its legality, and calculate the virtual address using the information inside `exec_env`. This way, the generated AOT code can perform address translation at runtime.

To support memory virtualization, we encode a module's two page tables into the `exec_env` structure and rewrite the address translation process using these page tables. Specifically, when compiling ordinary Wasm memory read (write) instructions (*i.e.*, `load` and `store` instructions), we inject IR instructions that extract the page index and offset from the Wasm address to be translated, look up the read (write) page table to obtain the virtual address of the corresponding Wasm page, and calculate the final virtual address by adding the page offset to the Wasm page's virtual address. For more complex memory instructions like `memory.copy`, `memory.fill`, and `memory.grow`, we insert IR instructions that invoke the runtime's functions to perform address translation and execute the corresponding operations.

B. Runtime Modification

The AOT-compiled native code is loaded and executed by the WAMR runtime, which we modify to support memory virtualization during execution. When instantiating a module, the runtime creates the necessary virtual pages with paddings, an exception page, and two page tables. When executing a function of a Wasm module, the runtime creates the `exec_env` structure with pointers to the page tables and passes it to the function. This allows the function to perform address translation using the page tables. We also implement a set of runtime functions to handle memory management requests from Wasm modules, corresponding to instructions such as `memory.grow`, `memory.size`, `memory.copy`, and `memory.fill`.

C. Runtime Interfaces

To support the evaluation of the data sharing capability of WAVEN (Sec. VI-G), we have implemented a preliminary set of runtime interfaces for shared memory.

User and module identification. We assume that the platform (*i.e.*, WAMR runtime) assigns unique user IDs (`userID`) to registered users (*i.e.*, Wasm module developer) and unique module IDs (`moduleID`) to the Wasm modules uploaded by users. `userID` and `moduleID` are integers that start from 0.

Shared memory management. Each shared memory region is identified by a unique string ID (`dataID`), which is assigned by users when creating the region. The runtime records the corresponding virtual pages for each shared memory region in a map (`shared_mem_map`) that maps the `dataID` to the virtual pages.

Authentication. To manage access permissions for shared memory regions, the runtime maintains a mapping (*i.e.*, `permission_map`) from the `dataID` to a `policy` structure. `policy` is a list of triples `(userID, moduleID, read_only)`, where `moduleID` records the module allowed to access the shared region, `userID` records the user who owns the module, and `read_only` is a boolean value indicating whether the module has write access to the shared region. Setting the `moduleID` or `userID` to -1 allows all modules or all users to access the shared memory region. For example, the policy triple `(-1, -1, false)` allows all modules to access the shared memory region with read-only permission, and the policy triple `(0, -1, true)` allows all modules uploaded by user 0 to access the shared memory region with write permission. If a module is not listed in the `policy` of a shared memory region, it is not allowed to access that region. The runtime performs authentication when a module attempts to access a shared memory region.

Interfaces. The `bool create_shared_mem(char* dataID, uint8_t* data, long len, policy* p)` interface allows a user's Wasm module to create a shared memory region. The module provides the shared data (`data`), the ID (`dataID`), the length (`len`), and the access policy (`p`) specifying which modules can access the shared data and their permissions. The runtime then updates `shared_mem_map` and `permission_map`, and returns `true` if the shared region is successfully created.

`uint8_t* access_shared_mem(char* dataID, long len)` interface allows modules to access shared data. The runtime first checks the existence of the shared data in `shared_mem_map`. Then, it verifies the access permission of the requesting module by consulting `permission_map`. If the shared data exists and the module has access, the runtime allocates a new region in the requesting module's memory space to store the data and performs memory remapping (see Sec. IV-E) to redirect the newly allocated region to the shared data with the specified access permissions (see Sec. IV-D).

The above interface implementation serves as a proof-of-concept, demonstrating that WAVEN can enable efficient data sharing with fine-grained access control across Wasm modules. Using WAVEN, platform owners can design more complex interfaces to support various secure data sharing scenarios in their platforms, such as self-expiring shared data.

VI. EVALUATION

In this section, we begin by presenting our experimental setup. We then evaluate WAVEN, examining the selection of padding sizes, the effectiveness of optimizations, and the performance in benchmarks, confidential computing workloads, memory-intensive operations, and memory sharing scenarios.

A. Experimental Setup

Testbeds. The testbed is a server that is equipped with 48-core Intel Xeon Gold 5318Y×2 processors (with SGX2 support) and 256GB of RAM. PVE 7.2 was installed on the host machine to create SGX2-compatible virtual machines. We perform the evaluation in a virtual machine with 16 vCPUs and 32GB of RAM. The virtual machine runs Ubuntu 20.04 with Linux kernel 5.15.0-70-generic and has 8GB of EPC.

Baseline. To validate the correctness and evaluate the performance of our implementation with WAVEN, we compare it with the vanilla WAMR implementation (commit `92c4bbe`). The WAMR implementation consists of an AOT compiler and a Wasm runtime. In generating SGX-compatible AOT code, the vanilla compiler inserts boundary check instructions for every memory access, since SGX does not support hardware-based boundary checks.

Benchmarks. We use PolyBench 4.2.1 [35] as our main test suite, which includes 30 numerical computation tasks. To perform memory stress test, we use STREAM [77], which measures the sustainable memory bandwidth of a system. We also consider two typical confidential computing workloads, to assess the practicality of WAVEN in real-world scenarios. Following the practice of the vanilla WAMR implementation, we compile the above benchmarks written in C into Wasm bytecode using WASI SDK 19.0 [73], and then into native code using WAMR's AOT compiler. We use the default optimization level (`-O3`) for both the clang compiler in WASI SDK and the AOT compiler. Each benchmark is executed with 128KB stack space and 256MB of heap memory, which is sufficient and will not trigger memory growth during execution. For PolyBench benchmarks, we use the default problem size (*i.e.*, large) throughout the evaluation.

B. Selection of Padding Size

The default page size of x86-64 Linux is 4KB and the cache line size of x86-64 processors is typically 64B. Although the Wasm page size is exactly 64KB, we need to pad each page with x bytes (see Sec. IV-C) to prevent the unsupported cross-page memory accesses from bypassing Wasm sandboxes. As a result, the padded Wasm page size is $64\text{KB} + x\text{B}$. The minimum value of x is 7B, which is sufficient for all Wasm memory access instructions, but it also makes the padded Wasm pages unaligned with system pages and cache lines.

To determine the optimal padding size, we measure the performance of our implementation by running PolyBench 50 times with different padding sizes: 0B (no padding), 7B (unaligned padding), 64B (cache line aligned padding) and 4096B (page aligned padding). Fig. 7 displays the results of the average execution time for each benchmark with the four padding sizes, showing that the performance is relatively stable across different padding sizes.

Compared to the baseline (no padding), the geometric mean of overheads is 4.56% for 7B padding, 5.4% for 64B padding, and 0.06% for 4096B padding. The similar performance of the 7B and 64B padding sizes suggests that cache line alignment is not a significant factor in padding size selection. Page-aligned padding introduces the minimal overhead, but consumes way more memory. Overall, 7B padding incurs insignificant overhead and has the smallest memory usage, indicating that it is a good trade-off between memory usage and performance. As such, we choose 7B as the padding size for the rest of the evaluation. With this setting, a Wasm module with 4GB memory (65536 pages) requires $65536 * 7\text{B} = 448\text{KB}$ of padding, which is deemed acceptable.

C. Effectiveness of Optimizations

1) Exception Pages and Page Paddings: By employing exception pages and page paddings, we achieve the same level of memory isolation as the vanilla linear memory model, but without requiring any boundary checks. Now, we demonstrate the effectiveness of this optimization by examining the overhead that would be introduced if we were to use checks. Table I presents the overheads incurred by inserting a boundary check to validate the target address in every memory access. The results show that there is a noticeable performance penalty when using boundary checks. Out of the 30 benchmarks tested, 21 of them experience more than 20% overhead. The maximum overhead is 64.69% (`doitgen`) whereas the geometric mean of overheads is 30.09%. These significant overheads highlight the effectiveness of the use of exception pages and page paddings.

Table I: Overheads of using boundary checks in PolyBench. Our implementation with exception pages and page paddings (thus no checks needed) is used as the baseline for comparison.

Benchmark	Overhead	Benchmark	Overhead	Benchmark	Overhead
2mm	54.07%	durbin	42.03%	lu	35.79%
3mm	55.30%	fdtd-2d	39.67%	ludcmp	10.51%
adi	31.38%	floyd-warshall	42.92%	mvt	22.06%
atax	58.87%	gemm	28.12%	nussinov	45.84%
bicg	22.73%	gemver	27.31%	seidel-2d	0.10%
cholesky	54.10%	gesummv	37.02%	symm	58.08%
correlation	3.44%	gramschmidt	13.04%	syr2k	11.92%
covariance	3.42%	heat-3d	28.63%	syrk	22.09%
deriche	8.06%	jacobi-1d	39.24%	trisolv	44.35%
doitgen	64.69%	jacobi-2d	17.33%	trmm	18.67%

2) Dual Page Tables: We leverage two separate page tables to enforce read-only memory semantics while avoiding permission checks. To evaluate the effectiveness of this design, we compare it with two alternative designs: 1) a scheme (referred to as `S-I`) using a single page table without permission checks, and 2) a scheme (referred to as `S-II`) using a single page table with permission checks for write instructions.

`S-I` is the fastest but lacks read-only memory semantics, and `S-II` checks each memory write to determine whether it is permitted. Compared to `S-I`, `S-II` exhibits a geometric mean overhead of 22.78% across PolyBench benchmarks, while the dual page table scheme only incurs an average overhead of 2.36%. Fig. 8 further displays the execution time comparison between the dual page table design and `S-II`, showing that our design performs much better without relying on heavy-weight permission checks (compared to our design, `S-II` is 19.96% slower in average). This comparison demonstrates that the dual page table design can achieve fine-grained memory access control with minimal overhead.

D. Performance

We first measure the performance of WAVEN in terms of compilation and execution, using the PolyBench benchmarks.

1) Compilation: In terms of compilation, we compile each PolyBench benchmark 50 times using both the modified AOT compiler and the vanilla AOT compiler and take the average compilation time for each. We observe that these

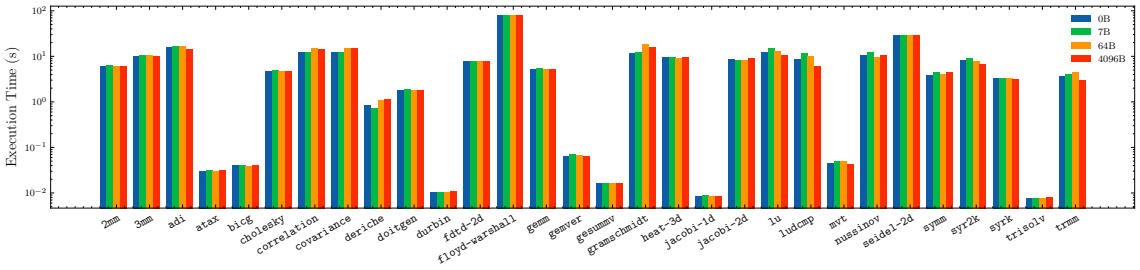


Figure 7: Execution time with different padding sizes (Log scale is used on the Y-axis).

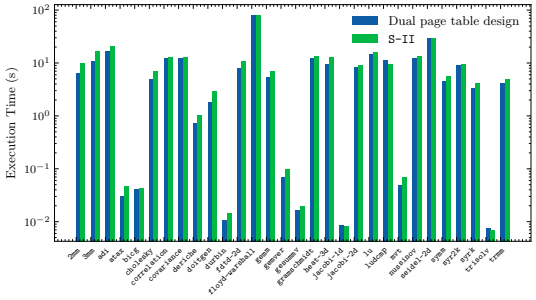


Figure 8: Execution time comparison: Dual page table design vs. `S-II` (Log scale is used on the Y-axis).

benchmarks have similar compilation time when compiled with the same compiler. The results, as shown in Table II, indicate that the compilation time of the modified compiler ranges from 1417ms (`gesummv`) to 1535ms (`ludcmp`), with an average value of 1457.07±29.02ms. On the other hand, the compilation time of the vanilla compiler ranges from 1491ms (`gesummv`) to 1600ms (`ludcmp`), with an average value of 1530.43±27.85ms.

Table II: Compilation time of the two compilers (unit: ms).

Value	Modified compiler	Vanilla compiler
Maximum	1535 (ludcmp)	1600 (ludcmp)
Minimum	1417 (gesummv)	1491 (gesummv)
Average	1457.07	1530.43
Standard deviation	29.02	27.85

When considering the size of the compiled code, the differences among benchmarks are not noticeable for the same compiler. As shown in Table III, the modified compiler generates code with an average size of 76.63 ± 0.84KB, whereas the vanilla compiler generates code with an average size of 89.73 ± 0.77KB.

In sum, the modified AOT compiler generates native code that is 16.9% smaller in size and has a 5.49% shorter compilation time compared to the vanilla compiler. This is expected since the modified compiler does not need to generate code for boundary checks, as the vanilla compiler does.

2) Execution: We execute the PolyBench benchmarks 50 times with our implementation and the vanilla WAMR implementation, respectively, to compare their execution time. The execution time is measured and printed by PolyBench itself, with the `POLYBENCH_TIME` macro enabled.

Table III: Sizes of the AOT-compiled code (unit: KB).

Value	Modified compiler	Vanilla compiler
Maximum	78	91
Minimum	75	89
Average	76.63	89.73
Standard deviation	0.84	0.77

Fig. 9 shows the results, including the average execution time of the 30 benchmarks in two runtimes (Fig. 9a), and the normalized execution time (our implementation vs. vanilla WAMR) (Fig. 9b).

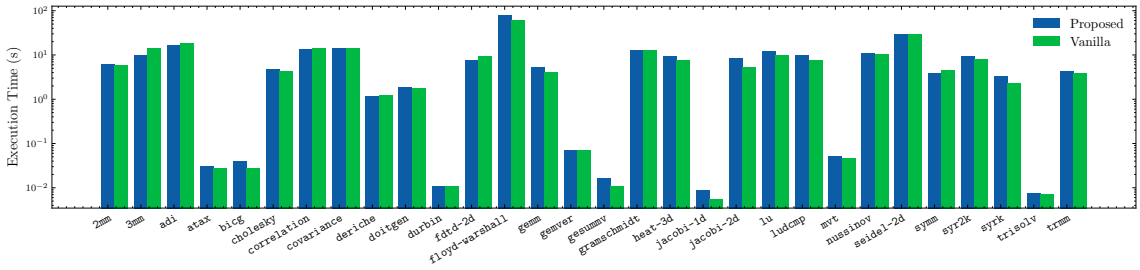
Compared to the linear memory model, WAVEN eliminates the need for boundary checks but introduces an additional memory read for every memory access due to the page table lookup in address translation. As illustrated in Fig. 9, the overall overhead of memory virtualization is relatively small, with the geometric mean of overheads reaching only 10.42%. The paging overheads are highly dependent on the memory access patterns of the benchmarks and tend to fluctuate. In some cases, memory virtualization introduces more overhead. For instance, the overheads of `jacobi-1d` and `jacobi-2d` are 54.61% and 56.50% (the maximum overhead), respectively. In other cases, memory virtualization even outperforms vanilla WAMR. For example, with `3mm`, our implementation performs 30.57% faster than vanilla WAMR.

E. Evaluation of Confidential Workloads

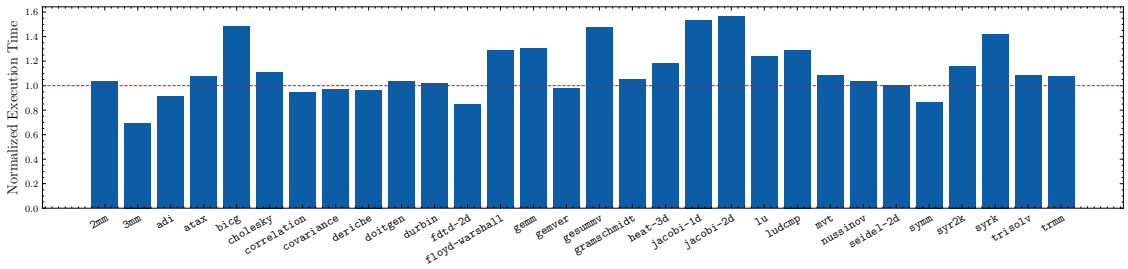
The result of micro benchmarks (Sec. VI-D) demonstrates the moderate overhead of WAVEN. To further assess its practicality, we also evaluate its performance in two commonly deployed confidential computing workloads, *i.e.*, confidential database and privacy-preserving machine learning.

Confidential database. We develop confidential in-memory databases by running an open-sourced Wasm-ported version [78] of SQLite v3.32.3 [79] in enclaves. `Speedtest1` [80] is a SQLite performance test program containing 32 tests, and the Wasm-ported code [78] consists of 29 tests. We run the ported tests in our implementation with WAVEN and the vanilla WAMR implementation 50 times, and measure the average execution time.

The 29 tests can be categorized into two groups, namely database query (14 tests) and database update (15 tests), and the overheads of our implementation in the two groups are displayed in Fig. 10. In query tests, the observed overheads vary between 1.92% and 23.09%, with a geometric mean of

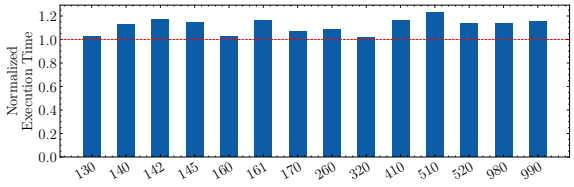


(a) Execution time (Log scale is used on the Y-axis).

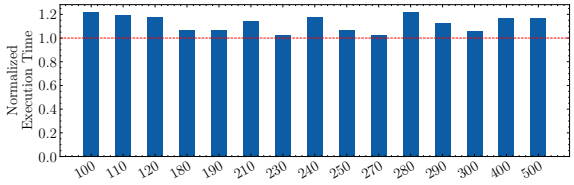


(b) Normalized execution time.

Figure 9: Execution performance in two implementations.



(a) Database query.



(b) Database update.

Figure 10: Overheads of WAVEN in SQLite tests. The X-axis shows the test ID in `Speedtest1`.

11.47%. Similarly, in update tests, the minimum and maximum overheads are 2.3% and 22.08%, respectively, with a geometric mean of 12.52%. The close overheads in the two groups indicate that WAVEN is stable for both read- and write-intensive database operations. Moreover, these figures prove that WAVEN introduces moderate overheads when deployed in confidential database workloads.

Privacy-preserving machine learning. Face detection is a representative privacy-preserving machine learning workload, where face images uploaded by end users should be kept confidential. We perform face detection in enclaves using a convolutional neural network (CNN) model [81], [82]. For this task, we utilize the WIDER FACE validation dataset [83], which contains 3,226 images categorized into 62 groups. Specifically, we select all 115 images from the first category (*i.e.*, Parade), as the input for the CNN model.

We measure the time required to process all 115 images and repeat the test 50 times. On average, our implementation with WAVEN takes 176.06 seconds to process the 115 images, while the vanilla WAMR implementation takes 165.87 seconds. This low overhead (6.14%) demonstrates that WAVEN is suitable for privacy-preserving machine learning workloads.

F. Memory Stress Test

To evaluate the performance of WAVEN under memory stress, we use STREAM [77], a well-known industry standard benchmark that measures memory bandwidths of different memory-intensive operations. As listed in Table IV, the benchmark consists of four kernels: Copy, Scale, Add, and Triad. Each kernel operates on large memory regions composed of three arrays. For instance, the Copy kernel copies all elements from array `b` to array `a`.

As requested in the benchmark website [84], each array must be at least 4× the size of the last-level cache used in the run. The default array size (*i.e.*, 10,000,000 elements per array, each element takes 8B and each array takes 76.3MB) satisfies this requirement (the last-level cache size of the testbed is 16MB) and is used in our evaluation.

In this memory stress test, we consider three implementations: 1) the vanilla WAMR implementation (`Vanilla`), 2) the vanilla WAMR implementation without boundary checks (`VanillaNoChk`), and 3) our implementation with memory virtualization (`Proposed`). `VanillaNoChk` disables boundary checks in the vanilla WAMR implementation and thus cannot provide memory isolation. Compared to `VanillaNoChk`, `Vanilla` inserts boundary checks per memory access, and `Proposed` uses memory virtualization to provide memory isolation. As such, `VanillaNoChk` is considered the baseline for comparison.

The reported bandwidths from STREAM is shown in Table IV, where we observe `Proposed` achieves more bandwidths than `Vanilla` in three of the four kernels.

The Copy kernel copying the entire array `b` to array `a` is compiled to one Wasm instruction (`memory.copy`). Thus, `Vanilla` only performs one boundary check for the entire array copy and has nearly the same bandwidth as `VanillaNoChk`. As explained in Sec. V-A–V-B, in WAMR implementation, the `memory.copy` instruction is finally handled by a runtime function that performs the actual memory copy. `VanillaNoChk` and `Vanilla` use linear memory model and have the same copy runtime function, which invokes `memmove` once to finish the copy. However, `Proposed` performs page-by-page copy and thus will invoke `memmove` many times for large copy operations, lowering the bandwidth.

For the remaining kernels, `Vanilla` achieves 64.17% (Scale), 62.94% (Add), and 63.15% (Triad) of the bandwidths of `VanillaNoChk`. In comparison, `Proposed` achieves 81.74%, 69.05%, and 67.6%, respectively. The overhead of `Vanilla` is attributed to boundary checks, while the overhead of `Proposed` is due to page table lookups. The better performance of `Proposed` compared to `Vanilla` in the Scale, Add, and Triad kernels illustrates that page table lookups have a lower overhead than boundary checks in memory stress operations, suggesting that WAVEN is suitable for memory-intensive workloads.

Table IV: STREAM benchmark details and evaluation results. The percentages in parentheses indicate the bandwidths relative to `VanillaNoChk`.

Kernel	Workload	Bandwidth (MB/s)		
		`VanillaNoChk`	`Vanilla`	`Proposed`
Copy	a[i] = b[i]	8215.7	8125.1 (98.9%)	6881.4 (83.76%)
Scale	a[i] = q*b[i]	4712.5	3023.8 (64.17%)	3851.8 (81.74%)
Add	a[i] = b[i] + c[i]	7050.8	4438.1 (62.94%)	4868.7 (69.05%)
Triad	a[i] = b[i] + q*c[i]	6998.3	4419.7 (63.15%)	4731.2 (67.6%)

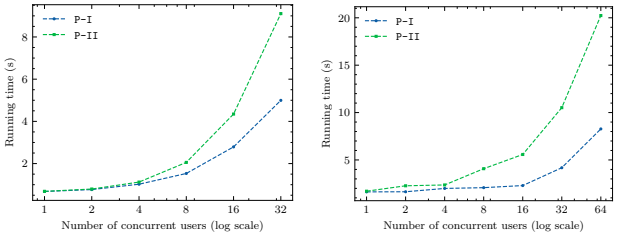
G. Effectiveness of Memory Sharing

As described in Sec. III, the need for data sharing with fine-grained access control is common in real-world confidential computing platforms, especially in confidential stateful FaaS and secure data marketplaces. To demonstrate the effectiveness of WAVEN in fulfilling this requirement, we develop two platform prototypes: one with memory virtualization (referred to as `P-I`) and the other without (referred to as `P-II`, which does not support shared memory). These two platforms are implemented based on the same enclave code, with the only difference being that `P-I` incorporates WAVEN while `P-II` employs the linear memory model. Then we evaluate the two platforms in two scenarios with different memory sharing patterns commonly encountered in confidential stateful FaaS and secure data marketplaces, and compare their performance.

1) Multi-write Multi-read Scenario: We first consider a typical scenario in confidential stateful FaaS, where a FaaS user creates a function chain consisting of multiple functions that read different parts of the same dataset, process the data in parallel, and then write the results back to the same place.

Specifically, we load the RCV1 dataset [85] (a collection of news stories from Reuters) downloaded from [86] into the enclave, with each copy of the dataset requiring around 900MB of memory. We then let each user train a text classifier using the same algorithm and dataset. Each user first creates a master function and then the master function will spawn 8 worker functions, each of which reads one-eighth of the dataset, computes the loss concurrently, and updates the central weights stored in the master function's memory. The master and worker functions implement the "HOGWILD!" algorithm [87], a lock-free stochastic gradient descent (SGD) algorithm designed for parallel computing. As such, different workers can update the same model weight vector stored in the master functions' memory space without synchronization.

In the evaluation, each FaaS function is a Wasm module, and we set the number of concurrent users to 1, 2, 4, 8, 16, and 32. With 32 users, the platform needs to execute 32 master modules and 256 worker modules in total. Fig. 11a shows the time taken to complete all users' tasks in the two platforms, and we observe that `P-I` gradually outperforms `P-II` as the number of concurrent users increases. When serving 16 and 32 concurrent users, `P-I` outperforms `P-II` by 1.56× and 1.82×, respectively. Memory sharing enhances the performance from two aspects: 1) it eliminates data copies from the original dataset to each user's worker functions, and 2) it allows workers to update the central model weights in situ. The above result shows that WAVEN can significantly boost the execution in multi-write multi-read scenarios.



(a) Multi-write multi-read scenario. (b) Multi-read scenario.

Figure 11: Evaluation of the effectiveness of memory sharing.

2) Multi-read Scenario: This scenario represents the memory sharing patterns in data marketplaces—multiple data buyers (users) can compute on the same purchased data inside the platform but cannot modify it.

We continue to use the RCV1 dataset and also let each data user train a classifier using the same code. However, in this case, each user only creates one Wasm module, which will read the entire dataset and finish training. The number of concurrent data users is set to 1, 2, 4, 8, 16, 32, and 64, respectively, and we also measure the total execution time required to complete all data users' tasks.

The results are plotted in Fig. 11b, where `P-II` exhibits a steeper slope compared to `P-I`. In highly-concurrent scenarios (*i.e.*, 16, 32, 64 concurrent data users), `P-I` outperforms `P-II`, achieving a speedup of 2.4×-2.5×. This substantial performance improvement is expected, as `P-II` copies the entire dataset for each user, resulting in both time and memory wastage. Notably, when running `P-II`, the evaluated enclave with a capacity of 8GB EPC can support a maximum of

seven concurrent users. Additionally, even when the number of concurrent data users is small (*e.g.*, 1, 2, 4), `P-I` still exhibits slightly better performance than `P-II`. These results demonstrate the effectiveness of WebAssembly memory virtualization in reducing memory utilization and enhancing performance in multi-read scenarios.

VII. Discussion

Prevelance of in-enclave WebAssembly. Compared to other SFI-based options, running Wasm to support in-enclave multi-tenancy is more appealing. First, Wasm is formally specified and verified. Wasm is specified in a formal semantics [16], which has been formally verified [17], [18], and the formal reasoning of real Wasm programs is also developed [88]. As such, incorporating Wasm into the TCB can be considered more reliable. Second, Wasm has a well-established ecosystem. Numerous programming languages can be compiled to Wasm, providing a wide range of options for developers. The Bytecode Alliance [19], which consists of industry leaders such as Amazon, Microsoft, and Intel, actively develops Wasm-based technologies. These advancements can also be leveraged for Wasm usage within enclaves.

Generalization to other TEEs. SGX is the most widely deployed user-space TEE and has been extensively adopted by academia and industry. Given that Intel will treat SGX as a first-class citizen in their confidential computing strategy [43], it is anticipated that WAVEN will be a very important addition in practice. Although our investigation and evaluation primarily concentrate on SGX, WAVEN can be applied to other user-space TEEs such as CURE [50] and SHELTER [51].

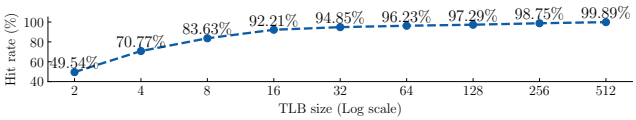
TLB implementation. The overhead introduced by memory virtualization is primarily due to the additional memory read during address translation. Implementing a software TLB may help amortize the overhead, and we report our TLB implementation and evaluation as follows.

Unlike hardware TLBs usually implemented in associative storages, most software TLBs use direct-mapped data structures to cache the translation results because of their simplicity and efficiency [89], [90]. Prominent examples include QEMU [91] and Spike [92], both of which use direct-mapped software TLBs. As analyzed by Hong *et al.* [89]: "SoftTLB is usually implemented as a directly mapped hash table relying on virtual guest addresses for efficiency. This is because, unlike the fully associative hardware TLB, SoftTLB cannot search its content in a parallel manner."

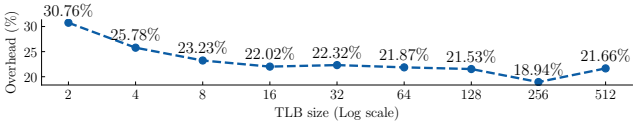
Following the common practice, we also implement a direct-mapped software TLB with 2, 4, 8, 16, 32, 64, 128, 256, and 512 entries. Each TLB entry caches a 64-bit virtual address for a Wasm page, and we use the modulo operation on the Wasm page index to locate the TLB entry. For instance, when the TLB size is 4, we use `Wasm_page_index % 4` to determine the TLB entry of a page. In the evaluation, to better observe the TLB performance, we configure both memory writes and reads to use a single page table. This setup, referred to as `S-I` in Sec. VI-C2, serves as the baseline. According to Sec. VI-C2, WAVEN and `S-I` exhibit similar performance. Therefore, the performance of `S-I` with TLB not only sheds light on the performance implications of TLB but also closely approximates the performance of WAVEN with TLB.

Fig. 12 displays the average TLB hit rate and performance overhead of different TLB sizes on PolyBench benchmarks. For each TLB size, the average TLB hit rate and overhead is obtained by computing the geometric mean of hit rates and overheads of all benchmarks. We observe that the performance overhead first decreases and then stabilizes, with the increase of the TLB size and the hit rate. When the TLB size is small, the hit rate is low, and the overhead is relatively high. For example, when the TLB size is 2, the hit rate is 49.54%, and the overhead is 30.76%. As the TLB size increases, the hit rate increases, and the overhead decreases and then stabilizes at around 21%. Even when the hit rate is 99.89% (512 TLB entries), the overhead is still 21.66%.

Our evaluation results in Sec. VI-C have shown that inserting checks in the memory access path causes significant overhead. Performing TLB lookups involves "miss or hit" checks and thus is expected to introduce similar overhead, which neutralizes the performance benefits of TLB. For the above analysis, we conclude that TLB cannot effectively amortize the overhead of memory virtualization in our implementation.



(a) Average hit rates



(b) Average overheads

Figure 12: TLB performance on PolyBench benchmarks.

VIII. Related Work

Intra-enclave isolation. Although SGX does not provide enclave isolation by default, there have been several proposed solutions to achieve this. Some of these solutions are based on hardware primitives such as Intel MPX [13] and Intel MPK [14], while others rely on software fault isolation. For instance, CHANCEL [8] proposes an SFI mechanism inspired by Native Client [24] to isolate multiple clients inside a single enclave. Each client's request is handled by a compartmentalized thread. However, CHANCEL lacks flexibility as it does not allow clients to execute customized code. Other solutions, such as MPTEE [9], Occlum [10], and Spons & Shields [11], utilize MPX to enforce enclave isolation. Unfortunately, MPX is immature [93] and later deprecated [94], which significantly limits the practicality of these prototypes. SGXJail [95] and SGXLock [96] leverage Intel MPK [14] to prevent potentially malicious enclaves from accessing arbitrary

host regions, but they do not achieve fine-grained (*i.e.*, intra-enclave) isolation. Moreover, as MPK demands trust on OS kernel, which contradicts to SGX's threat model, incorporating MPK inside enclaves to achieve intra-enclave is challenging and often leads to significant modifications that render the scheme less practical. For instance, LightEnclave [12] explores the combination of MPK and SGX to isolate multiple light-enclaves within an enclave, requiring not only MPK support but also hardware modifications, which reduce its practicality. In contrast, in-enclave WebAssembly, as demonstrated in this paper, efficiently supports intra-enclave isolation without relying on other hardware support or modifications, making it a better choice compared to the aforementioned approaches.

Fine-grained linear memory access control. Lehmann *et al.* [25] reveals that the lack of read-only memory semantics in Wasm's linear memory model can lead to security vulnerabilities, as a module's constant area can be overwritten by malicious inputs. To perform better access control for the linear memory, Lei *et al.* propose Domain Isolated Linear Memory (DILM) [97] model, where a Wasm module's memory is sliced into multiple data domains and each Wasm function has access to at most one domain. However, as it requires the use of MPK, DILM cannot work inside SGX enclaves. Moreover, from the perspective of data sharing, the DILM model focuses on memory protection within a single Wasm module; it does not specify how to support shared memory across modules. In contrast, WAVEN not only enables flexible memory sharing but also supports more fine-grained access control.

Comparison to MPK approaches. As mentioned above, Intel MPK has been used to enforce different isolation schemes in SGX enclaves [95], [96], [12] or to provide memory protection for Wasm modules [97]. We further compare WAVEN with these MPK-based approaches here.

SGXJail [95] and SGXLock [96] leverage MPK to confine enclave behaviors and protect the host OS, whose design goals differ significantly from ours. LightEnclave [12] uses MPK to achieve intra-enclave isolation, similar to WAVEN. LightEnclave requires hardware modification and is evaluated in an emulated setting, making it hard to compare LightEnclave with WAVEN directly. DILM [97] enables memory access control for multiple functions inside a single Wasm module using MPK, but it is incompatible with SGX's threat model due to its use of MPK. Besides, DILM is developed atop Wasmtime, which does not support SGX. As such, a direct performance comparison between WAVEN and DILM is difficult. As for developer effort, LightEnclave is built atop a library OS and supports native binary execution without modification, leading to a small developer effort. Both DILM and WAVEN require developers to incorporate interfaces into the Wasm module to utilize their features, resulting in a moderate developer effort. The above comparison is summarized in Table V. While we are unable to evaluate LightEnclave and DILM in the same environment as WAVEN, we list the performance overheads of LightEnclave and DILM from their respective papers.

Table V: Comparison to MPK-based approaches.

Approach	Overhead	Functionality	Deployment	Developer Effort
LightEnclave	4%	Intra-enclave isolation only	Hard	Small
DILM	10%	Separate data domains for different functions in a single Wasm module; Each domain has its own access control policy	Incompatible with SGX	Moderate
WAVEN	10.42%	Intra-enclave isolation with memory sharing across multiple Wasm modules and fine-grained memory access control	Easy	Moderate

WebAssembly use cases. Wasm is widely used in various use cases. For example, FaaS platforms like Fastly Compute Edge [98] and Cloudflare Workers [99] rely on Wasm sandboxes to execute untrusted user functions. Chadha *et al.* [100] leverage Wasm to host high-performance computing workloads, achieving near-native performance while reducing binary size. Internet of Things (IoT) runtimes like WAIT [101] and WiProg [102] facilitate IoT application development with the portability of Wasm. Furthermore, blockchain systems such as EOSIO [67], NEAR [68], and Ethereum [69] actively promote the use of Wasm for contract execution engines.

Confidential computing with WebAssembly. Wasm has also been widely adopted in confidential computing. TWINE [15] and AccTEE [20] both build two-way sandboxes by running a Wasm runtime in SGX. AccTEE also supports trusted resource accounting. Teaclave [3] and Se-Lambda [4] are two SGX-based confidential FaaS platforms that support Wasm function executors. Additionally, Wasm is used in other TEEs as well. WATZ [103] runs a Wasm runtime entirely in ARM TrustZone, providing a portable and secure execution environment for IoT devices. Enarx [23] is another Wasm-based confidential computing framework that supports multiple TEEs, including Intel SGX and AMD SEV. These works leverage Wasm's portability and security to enhance confidential computing. WAVEN, along with efficient data sharing and fine-grained memory access control, can further improve these platforms.

IX. Conclusion

We present WAVEN, an in-enclave memory virtualization scheme for WebAssembly, enabling efficient data sharing with fine-grained memory access control. Through evaluation in SGX enclaves, we demonstrate the efficiency and effectiveness of WAVEN. This work advances the field by addressing Wasm's memory limitations and facilitating secure and efficient data sharing in confidential computing environments.

Acknowledgment

The authors would like to thank the anonymous reviewers for the constructive feedback to this paper. Yinqian Zhang is in part supported by National Natural Science Foundation of china under grant No. 62361166633, National Key R&D Program of China under grant No. 2023YFB4503902, Shenzhen Science and Technology Program under grant No. JSGG20220831095603007, Futian District Trustworthy Confidential Computing Innovation Consortium, and a research grant from ByteDance Inc.

References

[1] "Intel Software Guard Extensions," https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html.

[2] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 857–874.

[3] "APACHE TEACLAVE," https://teaclave.apache.org.

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

[5] S. Zhao, P. Xu, G. Chen, M. Zhang, Y. Zhang, and Z. Lin, "Reusable Enclaves for Confidential Serverless Computing," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[6] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 533–549.

[7] W. Liu, W. Wang, H. Chen, X. Wang, Y. Lu, K. Chen, X. Wang, Q. Shen, Y. Chen, and H. Tang, "Practical and Efficient in-Enclave Verification of Privacy Compliance," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021, pp. 413–425.

[8] A. Ahmad, J. Kim, J. Seo, I. Shin, P. Fonseca, and B. Lee, "CHANCEL: Efficient Multi-client Isolation Under Adversarial Programs." in *NDSS*, 2021.

[9] W. Zhao, K. Lu, Y. Qi, and S. Qi, "MPTEE: Bringing Flexible and Efficient Memory Protection to Intel SGX," in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–15.

[10] 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 Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 955–970.

[11] 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*, 2021, pp. 186–200.

[12] 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 22)*, 2022, pp. 3129–3145.

[13] R. Ramakesavan, D. Zimmerman, and P. Singaravelu, "Intel Memory Protection Extensions (Intel MPX) Enabling Guide," *no. April*, 2015.

[14] "Memory Protection Keys," https://lwn.net/Articles/643797/.

[15] 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)*. IEEE, 2021, pp. 205–216.

[16] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 185–200.

[17] C. Watt, "Mechanising and verifying the WebAssembly specification," in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, ser. CPP 2018. New York, NY, USA: Association for Computing Machinery, 2018, pp. 53–65. [Online]. Available: https://doi.org/10.1145/3167082

[18] C. Watt, X. Rao, J. Pichon-Pharabod, M. Bodin, and P. Gardner, "Two Mechanisations of WebAssembly 1.0," in *Formal Methods*, M. Huisman, C. Păsăreanu, and N. Zhan, Eds. Cham: Springer International Publishing, 2021, pp. 61–79.

[19] "Bytecode Alliance," https://bytecodealliance.org.

[20] 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*, 2019, pp. 123–135.

[21] 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*, ser. CCSW'19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 185–199. [Online]. Available: https://doi.org/10.1145/3338466.3358916

[22] J. Ménétrey, M. Pasin, P. Felber, V. Schiavoni, G. Mazzeo, A. Hollum, and D. Vaydia, "A Comprehensive Trusted Runtime for WebAssembly with Intel SGX," *IEEE Transactions on Dependable and Secure Computing*, 2023.

[23] "Enarx," https://enarx.dev.

[24] 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," *Communications of the ACM*, vol. 53, no. 1, pp. 91–99, 2010.

[25] D. Lehmann, J. Kinder, and M. Pradel, "Everything Old is New Again: Binary Security of WebAssembly," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 217–234. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

[26] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful Functions-as-a-Service," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, 2020.

[27] R. Song, B. Xiao, Y. Song, S. Guo, and Y. Yang, "A Survey of Blockchain-based Schemes for Data Sharing and Exchange," *IEEE Transactions on Big Data*, 2023.

[28] "WebAssembly Documentation," https://webassembly.org/docs/security/.

[29] "WebAssembly Proposals," https://github.com/WebAssembly/proposals.

[30] "Multi-Memory Proposal for WebAssembly," https://github.com/WebAssembly/multi-memory.

[31] "How to write Rust Wasm code to take advantage of multi-memory," https://github.com/bytecodealliance/wasmtime/issues/4300.

[32] "Multi-memory support," https://github.com/WebAssembly/wasi-sdk/issues/211.

[33] S. Shillaker and P. Pietzuch, "Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 419–433. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shillaker

[34] "WebAssembly Micro Runtime," https://bytecodealliance.github.io/wamr.dev.

[35] "PolyBench/C 4.2.1," https://sourceforge.net/projects/polybench.

[36] "Wasmtime," https://wasmtime.dev.

[37] "Intel Trust Domain Extensions," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[38] "AMD Secure Encrypted Virtualization," https://www.amd.com/en/processors/amd-secure-encrypted-virtualization.

[39] "ARM Confidential Compute Architecture," https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture.

[40] "12th Generation Intel Core Processors Datasheet, Volume 1," https://www.intel.com/content/www/us/en/content-details/655258/content-details.html.

[41] "Support of AMD SEV-SNP in Amazon EC2," https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/sev-snp.html.

[42] "Azure Confidential VM Options," https://learn.microsoft.com/en-us/azure/confidential-computing/virtual-machine-solutions.

[43] "Rising to the Challenge-Data Security with Intel Confidential Computing," https://community.intel.com/t5/Blogs/Products-and-Solutions/Security/Rising-to-the-Challenge-Data-Security-with-Intel-Confidential/post/1353141.

[44] P.-C. Cheng, W. Ozga, E. Valdez, S. Ahmed, Z. Gu, H. Jamjoom, H. Franke, and J. Bottomley, "Intel TDX Demystified: A Top-Down Approach," 2023.

[45] "SGX Enclaves in Microsoft Azure," https://learn.microsoft.com/en-us/azure/confidential-computing/confidential-computing-enclaves.

[46] "SGX Enclaves in Alibaba Cloud," https://www.alibabacloud.com/help/en/ecs/user-guide/build-an-sgx-encrypted-computing-environment.

[47] "IBM Cloud Bare Metal Servers with SGX," https://cloud.ibm.com/docs/bare-metal?topic=bare-metal-bm-server-provision-sgx.

[48] "Intel Trust Domain Extension Guest Linux Kernel Hardening Strategy," https://intel.github.io/ccc-linux-guest-hardening-docs/security-spec.html#tdx-linux-guest-kernel-overall-hardening-methodology.

[49] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "SANCTUARY: ARMing TrustZone with User-space Enclaves." in *NDSS*, 2019.

[50] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf, "CURE: A security architecture with customizable and resilient enclaves," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1073–1090.

[51] Y. Zhang, Y. Hu, Z. Ning, F. Zhang, X. Luo, H. Huang, S. Yan, and Z. He, "SHELTER: Extending Arm CCA with Isolation in User Space," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[52] W. Wang, L. Song, B. Mei, S. Liu, S. Zhao, S. Yan, X. Wang, D. Meng, and R. Hou, "NestedSGX: Bootstrapping Trust to Enclaves within Confidential VMs," *arXiv preprint arXiv:2402.11438*, 2024.

[53] S. van Schaik, A. Seto, T. Yurek, A. Batori, B. AlBassam, C. Garman, D. Genkin, A. Miller, E. Ronen, and Y. Yarom, "SoK: SGX.Fail: How stuff get eXposed," 2022.

[54] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[55] "AWS Lambda," https://aws.amazon.com/lambda.

[56] "Google Cloud Functions," https://cloud.google.com/functions.

[57] Z. Jia and E. Witchel, "Boki: Stateful Serverless Computing with Shared Logs," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 691–707.

[58] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, and P. García-López, "Stateful Serverless Computing with CRUCIAL," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–38, 2022.

[59] "Open Data Documentation on AWS," https://github.com/awslabs/open-data-docs.git.

[60] "Data marketplace platform market report," https://www.imarcgroup.com/data-marketplace-platform-market.

[61] "Snowflake Marketplace," https://www.snowflake.com/en/data-cloud/marketplace/.

[62] "AWS Data Exchange," https://aws.amazon.com/data-exchange/.

[63] N. Hynes, D. Dao, D. Yan, R. Cheng, and D. Song, "A Demonstration of Sterling: A Privacy-Preserving Data Marketplace," *Proc. VLDB Endow.*, vol. 11, no. 12, pp. 2086–2089, aug 2018. [Online]. Available: https://doi.org/10.14778/3229863.3236266

[64] W. Dai, C. Dai, K.-K. R. Choo, C. Cui, D. Zou, and H. Jin, "SDTE: A Secure Blockchain-Based Data Trading Ecosystem," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 725–737, 2020.

[65] G. Su, W. Yang, Z. Luo, Y. Zhang, Z. Bai, and Y. Zhu, "BDTF: A Blockchain-Based Data Trading Framework with Trusted Execution Environment," in *2020 16th International Conference on Mobility, Sensing and Networking (MSN)*, 2020, pp. 92–97.

[66] "ETHEREUM VIRTUAL MACHINE (EVM)," https://ethereum.org/en/developers/docs/evm.

[67] "EOSIO Blockchain," https://eos.io.

[68] "NEAR Blockchain," https://near.org.

[69] "Ethereum Flavored WebAssembly," https://github.com/ewasm.

[70] "Multi-Memory Proposal for WebAssembly (The time when it was proposed)," https://github.com/WebAssembly/proposals/commit/af91cbf960d1c0bf820707d9eb299f395f942973.

[71] V. Costan and S. Devadas, "Intel SGX Explained," Cryptology ePrint Archive, Paper 2016/086, 2016, https://eprint.iacr.org/2016/086. [Online]. Available: https://eprint.iacr.org/2016/086

[72] "Emscripten," https://emscripten.org/index.html.

[73] "WASI SDK," https://github.com/WebAssembly/wasi-sdk.

[74] "RISC-V Instruction Set Manual," https://github.com/riscv/riscv-isa-manual.

[75] "ARMv5 Architecture Reference Manual," https://developer.arm.com/documentation/ddi0100/latest.

[76] "WebAssembly Micro Runtime Source Code," https://github.com/bytecodealliance/wasm-micro-runtime.

[77] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[78] "SQLite code ported to WebAssembly," https://github.com/JamesMenetrey/unine-twine/tree/main/benchmarks/sqlite.

[79] "SQLite website," https://www.sqlite.org/index.html.

[80] "SQLite Speedtest1 Program Description," https://sqlite.org/cpu.html.

[81] "YuNet: A Tiny Millisecond-level Face Detector (Source Code)," https://github.com/ShiqiYu/libfacedetection.git, (Accessed: 2022-05-03).

[82] W. Wu, H. Peng, and S. Yu, "YuNet: A Tiny Millisecond-level Face Detector," *Machine Intelligence Research*, pp. 1–10, 2023.

[83] "WIDER FACE: A Face Detection Benchmark," http://shuoyang1213.me/WIDERFACE/, (Accessed: 2022-05-03).

[84] "STREAM Benchmark Guidelines," https://www.cs.virginia.edu/stream/ref.html.

[85] D. D. Lewis, Y. Yang, T. Russell-Rose, and F. Li, "Rcv1: A new benchmark collection for text categorization research," *Journal of machine learning research*, vol. 5, no. Apr, pp. 361–397, 2004.

[86] "HOGWILD! Code Releases," http://i.stanford.edu/hazy/victor/Hogwild/.

[87] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," *Advances in neural information processing systems*, vol. 24, 2011.

[88] X. Rao, A. L. Georges, M. Legoupil, C. Watt, J. Pichon-Pharabod, P. Gardner, and L. Birkedal, "Iris-Wasm: Robust and Modular Verification of WebAssembly Programs," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, jun 2023. [Online]. Available: https://doi.org/10.1145/3591265

[89] D.-Y. Hong, C.-C. Hsu, C.-Y. Chou, W.-C. Hsu, P. Liu, and J.-J. Wu, "Optimizing control transfer and memory virtualization in full system emulators," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, dec 2015. [Online]. Available: https://doi.org/10.1145/2837027

[90] X. Guo and R. Mullins, "Fast TLB simulation for RISC-V systems," *arXiv preprint arXiv:1905.06825*, 2019.

[91] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX annual technical conference, FREENIX Track*, vol. 41, no. 46. California, USA, 2005, pp. 10–5555.

[92] "Spike RISC-V ISA Simulator," https://github.com/riscv-software-src/riscv-isa-sim.git.

[93] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2018.

[94] "Intel MPX Explained," https://intel-mpx.github.io.

[95] S. Weiser, L. Mayr, M. Schwarz, and D. Gruss, "SGXJail: Defeating Enclave Malware via Confinement," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019, pp. 353–366.

[96] Y. Chen, J. Li, G. Xu, Y. Zhou, Z. Wang, C. Wang, and K. Ren, "SGXLock: Towards Efficiently Establishing Mutual Distrust Between Host Application and Enclave for SGX," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4129–4146.

[97] H. Lei, Z. Zhang, S. Zhang, P. Jiang, Z. Zhong, N. He, D. Li, Y. Guo, and X. Chen, "Put Your Memory in Order: Efficient Domain-based Memory Isolation for WASM Applications," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 904–918.

[98] "Fastly Compute@Edge," https://docs.fastly.com/products/compute-at-edge.

[99] "Cloudflare Workers," https://developers.cloudflare.com/workers/platform/web-assembly.

[100] M. Chadha, N. Krueger, J. John, A. Jindal, M. Gerndt, and S. Benedict, "Exploring the Use of WebAssembly in HPC," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 92–106. [Online]. Available: https://doi.org/10.1145/3572848.3577436

[101] B. Li, H. Fan, Y. Gao, and W. Dong, "Bringing WebAssembly to Resource-Constrained IoT Devices for Seamless Device-Cloud Integration," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, ser. MobiSys '22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 261–272. [Online]. Available: https://doi.org/10.1145/3498361.3538922

[102] B. Li, W. Dong, and Y. Gao, "WiProg: A WebAssembly-based Approach to Integrated IoT Programming," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021, pp. 1–10.

[103] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "Watz: a Trusted WebAssembly runtime environment with remote attestation for TrustZone," in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2022, pp. 1177–1189.