WasmStone: Enabling Transparent Execution of Standalone WebAssembly Applications in Keystone

Quentin MICHAUD[1,2(✉)][0009−0007−0648−8966], Lukas HERTEL[3][0009−0007−8751−5478], Louis CAILLIOT[1][0009−0005−3518−3737], Dhouha AYED[1][0009−0005−0423−3375], Olivier LEVILLAIN[2][0000−0002−0558−5015], and Joaquin GARCIA-ALFARO[2][0000−0002−7453−4393]

[1] Thales Group, Palaiseau, France
[2] Télécom SudParis, Institut Polytechnique de Paris, Palaiseau, France
[3] Technical University of Munich, Munich, Germany

Abstract. Confidential computing is experiencing rapid adoption as organizations increasingly seek to protect sensitive data during processing in cloud and edge environments. As this paradigm matures, there is a growing need for transparent, portable, and auditable solutions. However, existing solutions are often tied to proprietary solutions and architectures, limiting transparency, portability, and security. In this paper, we present a novel secure abstraction layer that combines WebAssembly (Wasm) with the open source RISC-V Trusted Execution Environment (TEE) framework Keystone. Our approach embeds a Wasm runtime inside a Keystone enclave and introduces a multiplexed edge call mechanism to transparently bridge the WebAssembly System Interface (WASI) with the untrusted host operating system, while respecting Keystone's architectural constraints. This design enables unmodified Wasm applications to execute securely inside enclaves without requiring developers to write TEE-specific code. We detail the runtime adaptations required to support the latest version of WASI, discuss design trade-offs, and analyze security implications of host-mediated system calls. We evaluate our prototype using a representative Rust-based web application on both QEMU and a VisionFive 2 RISC-V board. Experimental results show an average overhead of approximately 10% compared to non-enclave execution. Our work illustrates that combining Wasm with an open RISC-V TEE provides a transparent, lightweight, and sovereign alternative to proprietary confidential computing solutions. Our complete solution and all code presented in this paper is reproducible and available as open source.

Keywords: Confidential Computing · WebAssembly · WebAssembly System Interface · RISC-V · Transparency

1 Introduction

Companies increasingly rely on public and hybrid cloud services. The share of enterprises using paid cloud computing services is particularly high among large enterprises where 84.67% reported purchasing such services in 2025, an increase of 6.9% compared with 2023 [12]. In such computing environments where the Cloud Service Provider (CSP) manages the whole software stack for client applications, data is left unprotected when being processed in memory.

Confidential computing aims to solve this challenge by using a hardware-based Trusted Execution Environment (TEE) [10]. Any software layer in-between the trusted hardware components and the running applications is excluded from the trusted environment. This way, sensitive data remains protected while being processed. Moreover, besides protecting data in use, confidential computing also provides integrity guarantees for processes running in the TEE. Recent regulations, such as the European Union Digital Operational Resilience Act (EU DORA) [13], now accelerate the shift of companies towards hybrid-solution or full adoption of confidential computing. According to a survey sponsored by the Confidential Computing Consortium [6], 18.3% of companies already deploy workloads in confidential computing production environments, and 56.7% are testing it. As adoption of confidential computing technologies grows, it is essential to provide solutions that are secure, transparent, portable, auditable, and easy to deploy.

However, confidential computing also has its downsides. Indeed, most confidential computing technologies today are based on proprietary hardware. This limits the auditability and portability of the TEEs, and lowers the confidence one can have in the hardware stack, when it is the most important component of a TEE. Furthermore, confidential computing only protects the guest application from the host machine, but does not protect the host against attacks coming from guest applications.

Another problem hindering the adoption of confidential computing is the difficulty of creating applications able to run on multiple TEEs. Indeed, most confidential computing solutions today requires writing TEE-specific code, restricting the adoption of confidential computing to a few use cases. Allowing applications to be executed transparently (i.e., without any modification) on a TEE is the logical next step for a broad adoption of confidential computing.

With the downsides listed above, future confidential computing solutions should take into account a threat model with two distinct attack scenarios. The first scenario is the classic threat model used for confidential computing, and implies a software attacker with total access to the host. The second scenario, less common but also critical for securing confidential computing environments, considers an attacker coming from a workload running inside a TEE, and targeting the host or another workload. In a survey, Michaud *et al.* [26] propose to leverage additional solutions such as WebAssembly (Wasm) and RISC-V, not only to improve the usability of confidential computing, but also increase its security, and therefore provide an answer to this threat model.

Wasm is a sandboxing technology and portable compilation target that can be leveraged to simplify the use of confidential computing solutions. By securing a Wasm runtime inside a TEE, any Wasm binary could run in the TEE without any modification. This approach paves the way for transparent execution of TEE-protected applications, helping popularize the adoption of confidential computing across a wide range of devices. Furthermore, leveraging Wasm inside a TEE offers a double sandboxing protection and answers the second scenario of the threat model. The sandboxing capability of Wasm ensures that each guest application is isolated from the host, and cannot interact with the host except through well-defined, secured interfaces.

To improve the auditability of the entire hardware stack, one solution is to leverage open Instruction Set Architectures (ISAs) and open hardware to design fully open confidential computing solutions. RISC-V is an open-licensed ISA that provides a great alternative to proprietary ISAs. It provides a great opportunity to build a transparent, portable, and auditable solution for confidential computing.

In this paper, we present WasmStone, a novel confidential computing solution that leverages Wasm and the open source RISC-V TEE framework Keystone to provide a transparent, portable, and auditable solution for executing applications without any modification on a TEE. First, we give the necessary background for understanding our approach and the limitations of existing solutions in Section 2. We then present the design of WasmStone in Section 3, before evaluating our solution in Section 4. Finally, we survey related work in Section 5 before concluding in Section 6.

2 Background

This section presents the necessary background for our solution. We start by presenting the confidential computing ecosystem, existing technologies, and challenges, before introducing Wasm and the WebAssembly System Interface (WASI).

2.1 Confidential Computing

Definition Confidential computing [10] is a security paradigm that aims to protect data in use by ensuring its confidentiality and integrity during computation, in addition to traditional protections for data at rest and in transit. Indeed, while conventional security mechanisms provide strong guarantees for data at rest (e.g., disk encryption) and data in transit (e.g., TLS), they offer limited protection for data during execution, when it must be available in plaintext in system memory. Confidential computing relies on hardware-enforced TEEs that isolate sensitive code and data from the rest of the system, including privileged software such as operating systems, hypervisors, and cloud administrators. An instance of an application running in a TEE is commonly called an enclave.

At the core of confidential computing is the assumption that parts of the software stack and system infrastructure cannot be fully trusted. To address this,

modern processors provide architectural mechanisms, such as memory encryption, isolated execution modes, and cryptographic attestation, that allow applications to execute within protected regions of memory. These regions ensure that sensitive workloads remain confidential even in the presence of a compromised or malicious host environment.

Formally, a confidential computing platform must guarantee that:

1. Data and code inside a trusted execution environment are protected from unauthorized observation or modification by external software.
2. The integrity of the execution environment can be remotely verified through cryptographic attestation.
3. Secrets are provisioned only after successful verification of the expected hardware and software state.

The confidential computing model shifts trust from the software stack to the underlying hardware and its security guarantees, thereby reducing the Trusted Computing Base (TCB) exposed to untrusted infrastructure.

Benefits of Open ISA Solutions for Confidential Computing Several hardware vendors have introduced confidential computing technologies that embody this paradigm. A description of the most important ones is provided in Table 1. Examples include x86-based solutions such as Intel Software Guard Extensions (SGX) or AMD Secure Encrypted Virtualization (SEV), ARM-based solutions such as TrustZone, and technologies based on other ISAs such as IBM Secure Execution (SE). Despite differences in design and threat models, these systems share a common goal: enabling secure computation in environments that cannot be fully trusted. A common point between these solutions is that they are proprietary and based on closed source hardware. This weakens the auditability and portability of the solutions for an essential component of confidential computing. Moreover, the French Cybersecurity Agency (ANSSI) stated in a technical report on confidential computing [15] that users must avoid running on shared infrastructure operated by providers they cannot trust, and are rather encouraged to leverage confidential computing to increase their security posture on dedicated hardware instead. In an ideal case, users provide dedicated hardware and choose the fitting TEE to run their workloads securely.

This is where open ISAs and open hardware bring interesting solutions: they increase the sovereignty of the confidential computing environments for clients by greatly decreasing the amount of hardware one needs to blindly trust. Furthermore, they allow building open confidential computing technologies that are flexible and auditable. This modularity makes it attractive for research, hardware prototyping, and security experimentation, since experts can inspect, modify, and formally verify the ISA without licensing constraints.

In this regard, RISC-V plays an important role. RISC-V is an open, modular ISA that follows an extensible design: a small base instruction set can be augmented with standard or custom extensions. Similarly to x86 privilege rings,

Table 1. Overview of selected Trusted Execution Environments.

Technology	ISA	Open solution
AMD Secure Encrypted Virtualization (SEV) [1]	x86	✗
Arm TrustZone [3]	ARM	✗
Arm Confidential Compute Architecture (CCA) [2]	ARM	✗
Confidential VM Extensions (CoVE) [29]	RISC-V	✓
IBM Protected Execution Facility (PEF) [17]	PowerPC	✓
IBM Secure Execution (SE) [5]	z/Architecture	✗
Intel Software Guard Extensions (SGX) [18]	x86	✗
Intel Trust Domain Extensions (TDX) [19]	x86	✗
Keystone [22]	RISC-V	✓
Penglai [14]	RISC-V	✓

RISC-V defines a hierarchical protection model composed of three primary privilege levels: Machine (M), Supervisor (S), and User (U) modes. Machine mode is the highest privilege level and has unrestricted access to hardware resources. It is typically used for boot firmware and low-level system control. Supervisor mode is intended for operating system kernels and manages virtual memory, interrupts, and system calls. User mode is the least privileged level and executes application code, relying on controlled traps into higher privilege levels to access system services. This structured separation enforces isolation between applications, the operating system, and firmware components, forming the basis for secure execution in RISC-V systems.

From a security perspective, RISC-V introduces various security mechanisms that can be leveraged to build confidential computing solutions. A key hardware mechanism in RISC-V is the Physical Memory Protection (PMP). PMP allows M-mode firmware to define fine-grained access control policies over physical memory regions, specifying read, write, and execute permissions for lower privilege levels (supervisor and user modes). Each PMP entry can enforce region-based isolation, enabling separation between operating systems, hypervisors, Security Monitors (SMs), and user applications. Several open confidential computing solutions have been built on top of RISC-V, including Keystone [22], Penglai [14] or CoVE [29].

Keystone We choose Keystone as the base TEE for our solution. The choice of Keystone is justified in Section 3, but the inner workings of Keystone are presented here.

Keystone is an open source framework for building customizable TEEs on RISC-V architecture, offering a great alternative to proprietary solutions. It requires the use of RISC-V's three privilege levels (U-mode, S-mode, and M-mode). The overall architecture of Keystone is shown in Figure 1.

Keystone relies on a small privileged component called the SM that runs at the highest privilege level (M-mode) and is responsible for enforcing enclave isolation. The SM constitutes the main part of Keystone's TCB and is the core component of Keystone's architecture. The main function of the SM is to manage

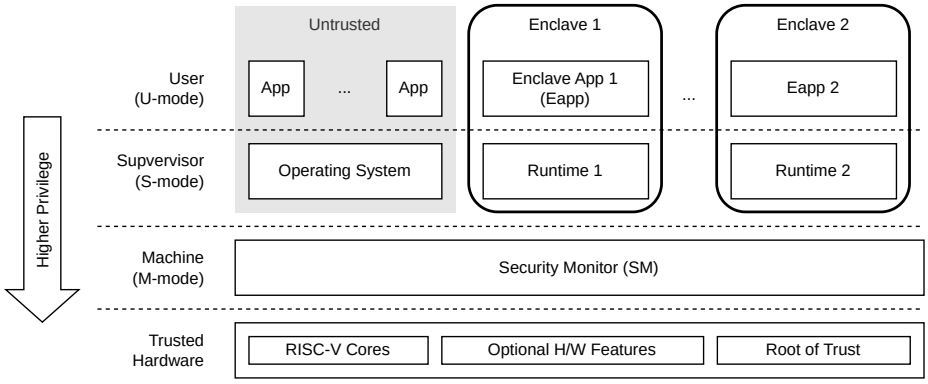

Fig. 1. Overview of the Keystone architecture

the PMP entries to partition physical memory and ensure that each enclave's memory region is accessible only to this enclave, while remaining protected from the operating system and other software. Inside the enclave, Keystone uses a runtime system that provides basic services normally handled by the operating system, such as memory management, handling the requests to and from the SM, and enclave lifecycle support. The default runtime of Keystone is Eyrie, a lightweight runtime designed specifically for Keystone enclaves, which implements a minimal set of abstractions and forwards necessary requests to the host system through controlled interfaces while preserving enclave isolation guarantees. This runtime runs in S-mode. Finally, the enclave application (eapp) runs on top of the runtime, leveraging its capabilities to execute code securely. This eapp runs in U-mode, the lowest privilege level. Keystone provide an SDK that allows developers to build their own enclaves using the Eyrie runtime, by easily writing code that leverage Eyrie capabilities.

Interaction with components outside the enclave (the untrusted host or other enclaves) occurs through a communication mechanism called edge calls (ecalls). This mechanism enables controlled transitions between trusted and untrusted execution contexts. In order to interact with the execution contexts on the untrusted host system, a companion component called the host application (happ) is necessary. The happ is charged to create the enclave and to serve as the mandatory communication bridge between the eapp and the outside world. This ecall communication mechanism between the eapp and the happ is illustrated in Figure 2.

When the eapp wants to access a resource on the untrusted host, it uses an ecall to exit the enclave (1), placing a marshalled[4] request containing the parameters in a shared buffer managed by the SM (2). The enclave stops its execution, giving control to the untrusted OS, and the happ asks the Keystone Linux driver located on the untrusted OS (3) to retrieve the content of the

[4] Converting a data field, or related structures, into a serialized string.

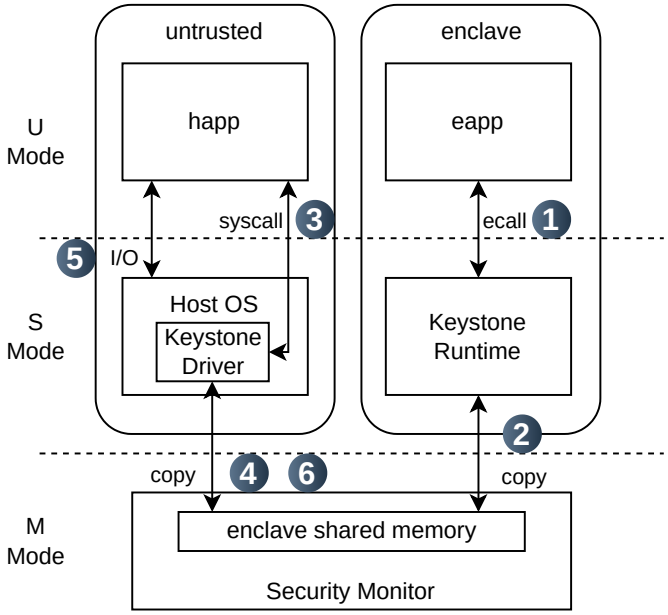

Fig. 2. Overview of the Keystone edge calls mechanism.

shared buffer from the SM (4). The happ then executes the required OS-level operations (such as I/O, networking, or file access) (5), writes the results back into the shared buffer (6), and let the eapp resume its execution.

2.2 WebAssembly and WASI

WebAssembly WebAssembly (Wasm) [28] is a low-level, platform-independent binary instruction format designed to serve as a portable compilation target for programming languages. Its primary goals are performance, portability, safety, and determinism across heterogeneous execution environments. Beyond being a binary format, Wasm defines a complete ISA implemented as a stack-based virtual machine, with security and performance as fundamental design objectives.

Although initially standardized for the web, Wasm has evolved into a general-purpose and cross-platform execution format. Today, Wasm runtimes exist for browsers, servers, embedded systems, and edge computing platforms. This has positioned Wasm as a promising foundation for portable software components, plugin systems, and secure multi-tenant execution environments.

A wide range of programming languages can be compiled to Wasm, supported by multiple compilers targeting diverse execution environments. Widely adopted source languages include C, C++, Rust, Go, and AssemblyScript, with continued expansion of language support. During compilation, toolchains typically embed additional runtime code and libraries to adapt the program to its target environment. For example, Wasm intended for execution in web browsers relies

on JavaScript bindings to access browser-provided features. As a consequence, such binaries are generally not portable to other contexts, such as standalone server-side execution, without recompilation or modification.

Wasm programs are executed by a stack-based virtual machine, where instructions consume operands by popping values from an evaluation stack and push their results back onto it. The architecture does not expose registers, relying entirely on stack-based operand passing. The Wasm bytecode itself resides in a memory region managed exclusively by the virtual machine; this code memory is read-only from the program's perspective and cannot be accessed or altered directly.

Wasm was designed with security as a core objective and executes code within a strictly isolated sandbox. A Wasm module has no implicit access to the host system and can only interact with external resources through explicitly provided interfaces exposed by the runtime. Its execution model enforces memory safety through bound-checked linear memory, structured control flow, and the absence of raw pointers or arbitrary jumps. Code, call stacks, and internal virtual machine structures are isolated from program data, significantly reducing the attack surface for classic memory corruption and control-flow hijacking attacks. This sandboxed design enables the safe execution of untrusted or third-party code across diverse environments while maintaining strong portability and predictable behavior.

WASI By design, Wasm does not natively expose interfaces to the host environment. Any interaction with external resources must occur through functions provided by the runtime, which mediate access to the host system and return results via linear memory or the evaluation stack. The availability and behavior of such functions are entirely runtime-dependent, motivating the need for standardization.

The WebAssembly System Interface (WASI) [8] is a standardized modular set of APIs (filesystem, sockets, clocks, etc.) that defines how Wasm modules interact with their host environment outside the browser. Each interface provides a set of functions that are implemented by the host runtime and are exposed to Wasm modules as part of the WASI runtime environment. While core Wasm deliberately omits any way of interacting with its outside environment, WASI provides these functionalities in a capability-based, sandboxed, and portable manner.

WASI decouples Wasm from web-specific APIs and enables Wasm programs to run as standalone applications on servers, directly on top of an operating system. Rather than exposing traditional system calls directly, WASI defines a narrow, explicit interface through which a host runtime selectively grants access to external resources such as files, directories, sockets, and environment variables.

A key architectural distinction of WASI is that it is not an operating system interface in the conventional sense. Instead, it is an abstraction layer that allows Wasm modules to remain portable across operating systems while relying on the host runtime to translate WASI calls into native system functionality. As a

result, the same Wasm binary can be executed unmodified across Linux, macOS, Windows, or embedded platforms, as long as a compatible WASI runtime is provided.

Although WASI continues to evolve, it has already seen widespread adoption. Two major versions are currently defined: WASI Preview 1 (WASIp1), released in late 2020, and WASI Preview 2 (WASIp2), released in early 2024. WASIp1 draws significant inspiration from the POSIX standards, but this design was clashing against the capability-based design of WASI and brought limitations to making Wasm a truly universal and modular application runtime. WASIp2 and future successors introduced a novel approach, the Component Model [7], to solve this problem.

3 Transparent WebAssembly Execution in a RISC-V TEE

In this section, we first present the objectives of our solution, WasmStone, and the design choices. We conclude with a discussion of the limitations of WasmStone and possible improvements. The full implementation of WasmStone is available as open source.[5]

3.1 Overview and Objectives

The goal of our solution is to enable the execution of Wasm code in a RISC-V enclave. The execution must be transparent: the exact same Wasm application must be able to execute seamlessly both inside and outside the enclave, and no TEE-specific code must be needed. Leveraging Wasm also allows us to improve the security of the solution by taking advantage of its sandboxing features to protect the host from a malicious or compromised guest.

Firstly, we must choose a RISC-V TEE solution that fits our needs. We aim to make our solution as broadly available as possible, so we target a TEE that leverages security features provided by default by the RISC-V ISA. We want an open source solution to be able to freely assess it and so that it can be easily inspected and modified if needed. Finally, we preferably look for projects that are still maintained and correctly documented. Several options are available, the most popular of them being listed in Section 2. CoVE [29] is the latest effort of the RISC-V community to build a secure TEE for RISC-V. However, standardization is still in progress, and no hardware is available yet for the solution. Penglai [14] is another interesting solution extending the RISC-V PMP to build flexible and secure TEEs for RISC-V. However, the extension of PMP is not standard and requires specific hardware. Finally, Keystone [22] is a popular solution for building a secure TEE for RISC-V. Implementations for QEMU and existing RISC-V hardware are available, and Keystone requires no hardware modifications. For these reasons, we choose Keystone as the base TEE of our solution.

[5] https://github.com/mh4ck-Thales/WasmStone

Table 2. Comparison of WebAssembly runtimes.

Runtime	Language	WASIp2	bare-metal RISC-V
Wasmtime[7]	Rust	●	◐
Wasmer[8]	Rust	○	○
WAMR[9]	C	○	●
WasmEdge[10]	C/C++	◐	○
Wasmi[11]	Rust	○	●

The first challenge of the solution design is to embed the Wasm runtime inside the Keystone enclave. Due to the constraints of Keystone design, this requires a carefully chosen runtime that validates several conditions. Next, to achieve full transparency, an implementation of WASI on top of the Eyrie runtime is needed, so that the Wasm application can freely interact with the host environment. This requires a well-thought design in order to work with the restrictions of Keystone regarding the communication with the untrusted host. Furthermore, the security guarantees of the enclave must be preserved even when allowing enclave application code to access the host environment.

3.2 Embedding WebAssembly Inside Keystone

Embedding a Wasm runtime inside a Keystone enclave is challenging. Firstly, the runtime must support compilation to bare-metal RISC-V, meaning it must not require any operating system dependencies (e.g., a standard library), and to be able to integrate with the constraints of the Keystone architecture. Secondly, given that upcoming WASI standards will evolve based on WASIp2, whereas WASIp1 will no longer be actively developed, selecting a runtime with support for WASIp2 is essential to ensure the long-term relevance and maintainability of our solution.

Several surveys assessing various Wasm runtimes have been published [34,33], allowing us to find the most relevant ones for our needs. These runtimes along with their support for the criteria listed above are listed in Table 2. After a careful comparison, we have opted for the Wasmtime runtime. Wasmtime fully supports WASIp2 and the Component Model, however its Just-In-Time (JIT) engine (Cranelift[6]) does not support running on bare-metal yet. We decided to use the slower Wasmtime interpreter (Pulley) instead.

After choosing the Wasm runtime, the next important design decision concerns where to position the Wasm runtime within the Keystone stack. Keystone is already based on a runtime, the default one being Eyrie. Any runtime compat-

[6] https://cranelift.dev
[7] https://github.com/bytecodealliance/wasmtime
[8] https://github.com/wasmerio/wasmer
[9] https://github.com/bytecodealliance/wasm-micro-runtime/
[10] https://github.com/WasmEdge/WasmEdge
[11] https://github.com/wasmi-labs/wasmi

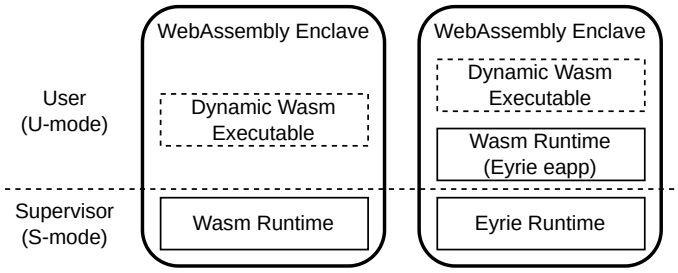

Fig. 3. Illustration of the two runtime approaches.

ible with the SM interface can be used. For example, the SeL4 microkernel [20], a formally verified OS kernel, has been successfully ported as a Keystone runtime.

A first possible approach is to implement our own Keystone runtime that also functions as a Wasm runtime. This would make our Wasm runtime run directly atop the SM, without the need for an intermediate layer. The advantage of this approach is that we can tailor the whole runtime to our needs, without having to accommodate the limitations or design decisions inherent in existing runtimes, such as performance impact or unneeded features. However, it would require to develop a complete Keystone runtime from scratch, including critical components such as memory management, attestation, and other security features. Since this runtime would run in the privileged S-mode of the RISC-V processor, it would also inherit high-level privileges, which conflicts with the principle of sandboxing.

A second possible approach is to reuse an existing runtime, such as Eyrie. The main advantage is to rely on an already proven to be secure and reliable runtime, while leveraging the features already implemented, such as the memory management. Finally, the Wasm runtime would be separated from the Keystone runtime and running in U-mode, the least-privileged mode used to restrict applications from modifying or observing certain system state, which sets up a better sandboxing. The Eyrie runtime also comes with an SDK that allows to easily write applications targeting it in C. The CONFIDENTIAL6G European research project[12] released another SDK[13] supporting Rust, called Vector SDK. The two approaches are illustrated in Figure 3. For our solution, we chose to write a Rust application on top of the Eyrie runtime to guarantee a strong security level and avoid the constraints of developing a new Keystone runtime.

Finally, the last challenge is to embed the chosen Wasm runtime in a Rust application on top of the Eyrie runtime. Since Eyrie provides a custom SDK, our Rust code requires the use of standalone libraries that support Rust's `no_std` feature, that is the ability to compile to bare-metal RISC-V without relying on a standard library. In the `no_std` environment, Rust still permits dynamic memory allocation if a custom memory allocator is provided. Libraries able to leverage custom memory allocators are directly calling the Rust `alloc` features. In our

[12] https://confidential6g.eu/
[13] https://github.com/vector-sdk/vector-keystone

application, we mapped the memory management features of Eyrie to the `alloc` features of Rust, in order to allow the application (and especially the Wasmtime runtime) to allocate memory. This mapping enables the application and, in particular, the Wasmtime runtime to perform memory allocations seamlessly within the constraints of the Eyrie environment.

3.3 Accessing Resources Outside of Keystone with WASI

WASI implementations must be provided to the Wasm runtime. When a Wasm runtime is used on a well-known OS, the WASI implementations are often provided directly by the Wasm runtime itself. However, when a Wasm runtime is used on a custom OS, it is up to the developers embedding the runtime to provide their own WASI implementation that Wasm applications will then use. For our solution, it means that WasmStone itself needs to provide WASI implementations that integrate0 with Keystone. Therefore, we need to write a WASI implementation to be used with the Eyrie runtime, leveraging Keystone ecalls to access resources on the untrusted host.

A first approach to implement WASI calls for workloads within a Keystone enclave is to assign one ecall for each WASI function. One WASI call is bound to one ecall using an ID, allowing the happ to know directly which action is required. To support the parameters of each function, small data structures are encoded and passed through the shared buffer of this same ecall. This way, the SM writes the arguments in the shared buffer and transfers the control to the host, then the happ reads the buffer containing the arguments and performs the required actions. Finally, the happ writes a result structure back in the shared buffer so that the eapp can read the result of the ecall.

However, Keystone's SDK builds a strong limitation for the maximum number of different ecalls that an enclave can create by fixing a limit of 10 ecall slots. While this limitation could be increased by modifying the source code of Keystone, we choose to avoid any modification to Keystone for the sake of simplicity and portability of our solution. With each distinct WASI call consuming one slot, it represents an important limitation of the one-to-one approach, making the implementation of more than 10 WASI calls impossible.

A solution to overcome this limitation is to multiplex multiple WASI calls within a single ecall. Therefore, the eapp and happ must agree on a small, well-defined protocol that tells the host which WASI call is being requested and carries the needed arguments, as depicted in Figure 4 and described below.

The rationale behind the multiplexing approach is to use one ecall for each WASI interface (e.g., filesystem, sockets, clocks), so each interface gets its own entry in the ecall table. For each ecall, a WASI function ID is encoded on one byte, telling the happ which WASI function of that interface to execute on the untrusted host. This allows us to implement 10 WASI interfaces with a maximum of 256 calls per interface, which is enough for supporting the complete WASIp2 standard. However, to support more WASI calls in the future, the one-byte encoded function ID location in the ecall can be increased to support a larger number of WASI functions. As long as this implementation parameter is shared

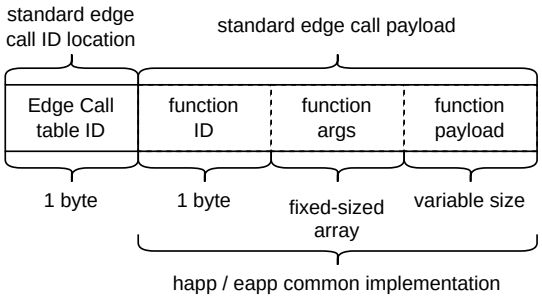

Fig. 4. Edge call structure following the multiplexing approach.

between the eapp and the happ, the solution can be adapted to support more than 256 WASI calls per ecall slot if needed.

To pass the function arguments to the happ, the data structure coming right after the function ID sets the required parameters to execute the function (e.g., path, flags, or mode for a filesystem function). This data structure size is fixed by the implementation for a given WASI call and shared between the eapp and the happ. After the fixed-size structure comes a buffer used to handle the potential payloads of the function whose arguments are not fixed-size (e.g., data to write inside a file of the host). The data structure and protocol definition shared between the happ and eapp guarantee the protocol compatibility. The multiplexing approach therefore allows us to drastically increase the number of WASI calls available for an enclave, without increasing the number of ecall slots.

3.4 Limitations and Possible Improvements

In terms of security, the current implementation of the WASI calls leaves the data sent between the eapp and the external resources unencrypted and unauthenticated by default. The happ is untrusted as it is running on the untrusted host. In this situation, a malicious host could intercept or modify communications between the eapp and external resources (e.g., filesystem, network, and randomness). If this data is sensitive, this could be used to compromise the enclave's security. For operations such as requesting random data for cryptographic purposes, the security of communications between external resources and the eapp is paramount.

As an example, the current design treats the `random_get` WASI call like any other I/O operation: the enclave asks the host for a given content, the host fetches the content and fills the shared buffer with it, and the enclave reads the bytes back. For obtaining secure randomness with the `random_get` WASI call, it is the untrusted host that retrieve random data and copy it to the shared buffer. Because the communication with the randomness source is not encrypted or authenticated, any untrusted component can observe or alter the random bytes shared between the happ and the entropy source. This defeats the confidentiality guarantees of the enclave for any operation that relies on fresh secrets (e.g.,

key generation, nonce creation). To preserve secrecy, the WASI call must be wrapped in an additional secure envelope (e.g., an authenticated encryption scheme), which adds latency and requires key management between the enclave and the external resources. Furthermore, the randomness source would need to be trusted; it is often managed by the host OS itself, which is untrusted in this scenario. Bringing such security into this solution would require a more complex design that is not covered in this paper.

Another solution would be to handle the WASI random interface only within the enclave. However, the enclave itself does not have an internal mechanism to generate secure pseudo-randomness, and needs to retrieve it from an external component. This limits the possibility of implementing some WASI standards, such as the random interface.

However, this does not compromise the enclave security for non security-related operations. The goal of the TEE is to protect data in use, and the application chooses what goes outside the enclave through the WASI calls. The interactions of a malicious host could result in a denial of service, but this is not taken into account by confidential computing's threat model. For simpler usages, such as writing logs or using unencrypted networking, this approach is sufficient.

The implementation we propose is an initial proof of concept for our design that can be further adapted to adopt production best practices. More precisely, the solution is built around the latest WASIp2 specification which is still evolving. API names, data representations, and the set of supported interfaces may change in further previews. To minimize these risks, we have designed the code to be easily adaptable to new WASI standards and future versions.

The Pulley Wasmtime interpreter represents another limitation of our solution, as the JIT engine is not yet available for bare-metal compilation. However, this limitation not only concerns our solution but all the others targeting the use of WASIp2 and RISC-V. Furthermore, as the Wasmtime developers are working on making the JIT engine available for bare-metal compilation, we hope to see this limitation disappear in the future. We are confident that our solution can be adapted to the Wasmtime JIT engine compiled to bare-metal once it becomes available, thus improving portability, transparency, and performance.

4 Evaluation

In this section, we present our evaluation of the proposed solution, using a realistic application as an example.

4.1 Application Used for Evaluation

We evaluate our solution using a compact realistic application written in Rust, which implements an image classification task served through a website. The

classifier is based on the MNIST database of handwritten digits [21]. The code of this application is also made open source.[14]

This application is developed without any consideration for execution inside an enclave. As mentioned in the previous section, we need to compile our application to the Pulley format before executing it inside the enclave. To ensure a relevant comparison, we evaluate the exact same Pulley-compiled application both inside and outside the enclave. This application can then be executed using the Wasmtime runtime.

While RISC-V support the faster Cranelift JIT engine, we aim to compare the difference in performance brought by our solution in similar execution environments. When compiling to the Pulley format, we aim to modify the application as less as possible.

We compare minimally modified binaries, modifying only what is strictly necessary to run the application in an enclave. The only modification is constraining the maximum memory size to allow the Wasm application to run within the enclave's limited memory. Indeed, by default, Wasmtime tries to allocate 4 GB of linear memory at the start (the maximum size for a 32-bit Wasm module). This large allocation fails in the enclave that does not have enough memory. We therefore limit the memory size to 256 MB, which is enough for small applications being deployed on IoT devices.

4.2 Evaluation Targets

Our solution is both evaluated on the QEMU virtual machine provided by the Keystone project (executed on an x86 host), and on real RISC-V hardware, the VisionFive 2 (VF2). The VF2 is a RISC-V single-board computer featuring a quad-core 64-bit CPU and 8 GB of RAM. It provides upstream Linux support on actual RISC-V hardware, and is easily available, making it ideal for this evaluation. To ensure the fairness of the evaluation, we use the same hardware configuration for both QEMU and the VF2 (4 cores and 8 GB of RAM). The evaluation is performed using the Wasmtime runtime version 33.0.1, as it is the version currently used in WasmStone. In order to assess the overhead, we compare the performance of the application when running inside and outside the enclave (directly on the untrusted Linux host). The Linux hosts used in this assessment are built with the Buildroot project,[15] a cross-compilation toolchain for embedded Linux systems. Buildroot is used by the Keystone project to build the Linux system used by Keystone. For executing the application outside the enclave, the official GitHub release of Wasmtime for the corresponding system is used.[16]

[14] https://github.com/mh4ck-Thales/wasmstone-demo-app
[15] https://buildroot.org
[16] https://github.com/bytecodealliance/wasmtime/releases/tag/v33.0.1

4.3 Results

WasmStone is composed of only 3100 lines of Rust code for the part sitting within the enclave, and of 900 lines of Rust code for the part sitting on the host. That makes it a fairly small solution and minimizes the addition of TCB.

We first assess if the solution is able to run the application without errors. Enabling the logs in the enclave shows that the application is leveraging several WASI calls (filesystem, clocks, TCP sockets, I/O) without issues. Overall, the application performs normally when running outside the enclave and runs flawlessly inside, demonstrating the transparency and effectiveness of our solution. It should be noted that this application is relatively simple; more complex applications may encounter corner cases since not all WASI calls are supported, and some were implemented using workarounds and do not perfectly respect the WASI specification. Nevertheless, if this proof of concept is extended with robust, standard-compliant WASI implementations, it should operate flawlessly with any kind of application.

Next, we evaluate the performance of our application on the targets described in Section 4.2. For the evaluation, we sequentially send HTTP requests to the application, each containing a randomly selected image from the MNIST testing dataset [21] on which the classifier was trained. This process is repeated 100 times, and we measure the total execution time. Each target is tested 10 times. The evaluation code and scripts used for generating related figures are available as open source.[17] The results are presented in Figure 5.

As expected, executing the application within the enclave introduces performance overhead. This overhead is already present in the baseline Keystone solution [22], and is likely increased by our approach due to frequent interactions with the untrusted host. Nevertheless, the measured overhead of 24% for the QEMU target and under 10% for the VF2 target is reasonable given the security guarantees the enclave provides.

The notable difference in overhead between QEMU and VF2 can be explained by Keystone's enclave protection mechanisms (e.g., PMP management) being more costly under emulation. This reinforces the assertion that the overhead around 10% on real hardware is a fair trade-off for enhanced security, even for production environments.

We also notice that the results are fairly more dispersed for QEMU experiments than for experiments on the VF2. This can be explained by the fact that QEMU is a VM running on a host that also runs other processes, thus varying the response time. On the other hand, the VF2 is only running the enclave and without a virtualization layer, therefore providing with more reliable execution times.

Interestingly, the application running in QEMU outperforms the one running on VF2 hardware. This discrepancy likely stems from the early development stage of RISC-V CPUs; thus, emulation on a mature x86 host can be inherently faster than execution on emerging RISC-V platforms.

[17] https://github.com/mh4ck-Thales/wasmstone-eval

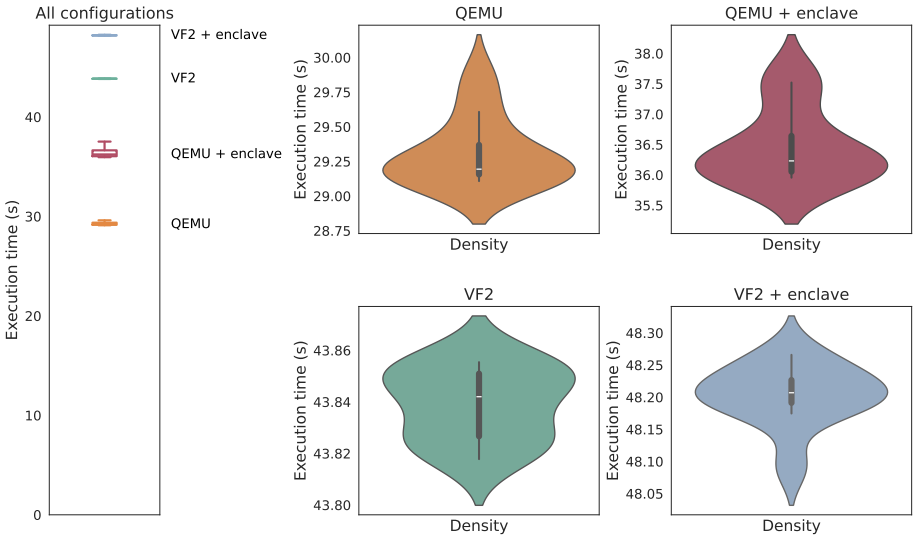

Fig. 5. Performance comparison of the WasmStone evaluation application in several situations.

5 Related Work

Several approaches to ease the deployment of confidential computing workloads already exist. In general, they aim at allowing application developers to leverage confidential computing as broadly and efficiently as possible. Michaud *et al.* [26] provide a survey of such existing technologies, which they call abstraction layers. Two main approaches are identified, container-based and Wasm-based. Container-based approaches are more flexible but less lightweight and portable than their Wasm-based counterparts.

Container-based approaches have been proposed to facilitate the execution of applications within trusted hardware environments. For instance, SCONE (Secure CONtainer Environment), by Arnautov *et al.* [4] enables containerized workloads to execute inside Intel SGX enclaves while requiring only limited modifications to existing applications. Its design centers on embedding a tailored C standard library within the enclave, in order to provide transparent execution. This library becomes part of the TCB, allowing applications to operate largely unchanged. Similarly, Gramine, by Tsai *et al.* [31] is a lightweight library operating system that supports the execution of unmodified Linux binaries within trusted execution environments. It offers a compact, kernel-like abstraction layer that reproduces essential Linux system interfaces inside the enclave. Beyond native SGX support, Gramine provides an auxiliary component known as the Gramine Shielded Container (GSC), which integrates with container tooling to transform standard OCI-compatible images into enclave-protected containers. Finally, Confidential Containers (CoCo), by Valdez *et al.* [32] extends confiden-

tial computing to cloud-native infrastructures. The project enables Kubernetes-managed workloads to benefit from hardware-enforced isolation guarantees. Its architecture encapsulates each container within a lightweight virtual machine backed by a trusted execution environment. A primary objective is to conceal heterogeneity across different TEE technologies, allowing developers to deploy confidential applications without platform-specific adaptations. Current support includes Intel SGX and TDX, AMD SEV, and IBM SE.

Within the Wasm ecosystem, several projects aim to combine Wasm portability with hardware-backed isolation. Enarx [11] is an open source initiative that delivers a trusted runtime for Wasm applications, currently targeting Intel SGX and AMD SEV. It relies on WASI and Wasmtime to expose standardized system interfaces required for secure execution and communication. This design promotes cross-platform portability while preserving the security guarantees offered by the underlying TEE.

Ménétrey *et al.* introduce TWINE (Trusted Wasm IN Enclave) [23], a compact and embeddable Wasm virtual machine tailored for Intel SGX, later complemented by a detailed performance evaluation [25]. TWINE builds upon WAMR (WebAssembly Micro Runtime), selected for its native support for SGX. WAMR provides an ahead-of-time (AOT) compilation toolchain that leverages LLVM to translate Wasm bytecode into native binaries prior to enclave deployment. Consequently, TWINE omits an interpreter and exclusively executes AOT-generated code. This architectural choice yields performance benefits over interpretation and reduces memory consumption, an important constraint in SGX and resource-limited cloud or edge environments. TWINE's contributions have since been merged into the upstream WAMR project. Extending their work to other TEEs, Ménétrey *et al.* propose WaTZ (Wasm in TrustZone) [24], an open source runtime designed for secure execution of Wasm modules within Arm TrustZone. Although developed independently to accommodate the architectural specificities of TrustZone, WaTZ adopts several principles from TWINE, including the use of WASI, integration with WAMR, and reliance on the AOT execution model. The system specifically targets constrained Arm-based edge devices.

Other efforts explore alternative runtime stacks. Se-Lambda by Qiang *et al.* [27] extends OpenLambda to support Intel SGX-based isolation. AccTEE by Goltzsche *et al.* [16] provides a trusted execution environment built around the Node.js ecosystem, employing Intel SGX as its hardware root of trust. To support both JavaScript and Wasm workloads, AccTEE leverages V8, the JavaScript engine developed for the Chrome browser. Finally, Veracruz [9] offers a framework for privacy-sensitive applications using a Wasm-based trusted runtime. It also adopts WASI to supply standardized system services, such as access to an in-memory file system and cryptographic randomness. For execution, Veracruz integrates Wasmtime as its underlying Wasm engine, combining portability with TEE-enforced confidentiality guarantees. Finally, the CONFIDENTIAL6G project released a proof of concept of Wasm running inside their Vector SDK [30], but this proof of concept uses a deprecated Wasm runtime (TinyWasm) and does not support WASI.

Table 3. Comparison of WasmStone and previous work

Existing work	Approach	Target ISAs	Open source	WASIp2 support
CoCo [32]	Container	x86, z/Architecture	✓	N/A
Gramine [31]	Container	x86	✓	N/A
SCONE [4]	Container	x86	✗	N/A
AccTEE [16]	Wasm	x86	✓	✗
Enarx [11]	Wasm	x86	✓	✗
Se-Lambda [27]	Wasm	x86	✗	✗
TWINE [23]	Wasm	x86	✓	✗
Vector [30]	Wasm	RISC-V	✓	✗
Veracruz [9]	Wasm	ARM	✓	✗
WaTZ [24]	Wasm	ARM	✓	✗
WasmStone	Wasm	RISC-V	✓	✓

From Table 3, we notice that only one other existing work tackles RISC-V TEEs. This highlights our solution capability to leverage TEE technologies that are more verifiable than their proprietary counterparts. Additionally, WasmStone is the only work that supports the latest WASIp2 specification.

Furthermore, our solution is implementing WASI in a completely transparent way and is able to execute the exact same Wasm application both inside and outside an enclave. This is not the case for most of the existing works, which hamper the transparency and easiness of integration.

6 Conclusion

We have introduced WasmStone, a confidential computing architecture that enables transparent execution of standalone Wasm applications inside a Keystone enclave. WasmStone is the first solution enabling transparent WASI-based execution of Wasm applications in a RISC-V trusted execution environment. We successfully reduced the complexity of confidential computing adoption by allowing Wasm binaries to run inside a TEE without requiring developers to write TEE-specific code, while improving its security.

Our design combines hardware-enforced enclave isolation with Wasm's sandboxing guarantees, strengthening the security guarantees of confidential computing with two-way sandboxing. We therefore not only protect against attacks from the host targeting guest applications running in TEEs, but also against attacks from a compromised or malicious guest application targeting the host or other workloads. A key contribution of our work is the implementation of transparent WASI support inside a Keystone enclave. We designed a transparent WASI layer that bridges the enclave application to the untrusted host through a multiplexed ecall mechanism compatible with Keystone's architectural constraints. This enables the same Wasm application to run seamlessly both inside and outside the enclave. Our evaluation shows that the system can run realistic workloads with modest overheads, less than 10% on real hardware, demonstrating the practicality of this approach.

While our prototype highlights some implementation challenges, most limitations are tied to the maturity of the surrounding ecosystem. In particular, the current need for precompiling the Wasm binary before running it inside the enclave is expected to disappear once the chosen Wasm runtime supports the required features, enabling unmodified execution of Wasm binaries.

Overall, WasmStone illustrates that combining Wasm with open RISC-V TEEs provides a transparent, portable, and auditable approach to confidential computing. Compared to existing approaches, WasmStone fills an important gap in the confidential computing ecosystem with unparalleled advances in transparency on RISC-V hardware, thus opening the door to developer-friendly and sovereign confidential computing platforms. As future work, our solution can be further improved by designing secure communication protocols with external components to better integrate security-related WASI interfaces to WasmStone.

Acknowledgments. This research received funding from the Smart Networks and Services Joint Undertaking (SNS JU) under the European Union's Horizon Europe programme: ELASTIC (GA 101139067). The authors have no competing interests to declare that are relevant to the content of this article.

References

1. AMD: Secure Encrypted Virtualization (SEV), `https://www.amd.com/en/developer/sev.html`, published: 2016; Last access: March 2026
2. Arm: Confidential Compute Architecture, `https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture`, published online: 2021; Last access: March 2026
3. Arm: TrustZone for Cortex-A, `https://www.arm.com/technologies/trustzone-for-cortex-a`, published online: 2002; Last access: March 2026
4. Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M.L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., Fetzer, C.: SCONE: Secure linux containers with intel SGX. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). pp. 689–703. USENIX Association, Savannah, GA (Nov 2016), `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov`
5. Bornträger, C., Bradbury, J.D., Bündgen, R., Busaba, F., Heller, L.C., Mihajlovski, V.: Secure your cloud workloads with IBM Secure Execution for Linux on IBM z15 and LinuxONE III. IBM Journal of Research and Development **64**(5/6), 2:1–2:11 (2020). `https://doi.org/10.1147/JRD.2020.3008109`
6. Bues, P.: Unlocking the future of data security: Confidential computing as a strategic imperative, `https://confidentialcomputing.io/wp-content/uploads/sites/10/2025/11/US53866125.pdf`, published online: Nov. 2025; Last access: March 2026
7. Bytecode Alliance: The WebAssembly Component Model, `https://component-model.bytecodealliance.org/`, last access: March 2026
8. Clark, L.: Standardizing WASI: A system interface to run WebAssembly outside the web - Mozilla Hacks – the Web developer blog, `https://hacks.mozilla.org/`

2019/03/standardizing-wasi-a-webassembly-system-interface, published online: March 2019; Last access: March 2026
9. Confidential Computing Consortium: https://veracruz-project.com, last access: March 2026
10. Confidential Computing Consortium: A technical analysis of Confidential Computing v1.3, https://confidentialcomputing.io/wp-content/uploads/sites/10/2023/03/CCC-A-Technical-Analysis-of-Confidential-Computing-v1.3_unlocked.pdf, published online: Nov. 2023; Last access: March 2026
11. Confidential Computing Consortium: Enarx, Confidential Computing with WebAssembly, https://enarx.dev, published: 2019; Last access: March 2026
12. European Commission, Eurostat: Cloud computing – statistics on the use by enterprises, https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Cloud_computing_-_statistics_on_the_use_by_enterprises, published online: Jan. 2026; Last access: March 2026
13. European Parliament and Council of the European Union: Digital Operational Resilience Act: regulation (EU) 2022/2554 of the European Parliament and of the Council of 14 December 2022 on digital operational resilience for the financial sector and amending Regulations (EC) No 1060/2009, (EU) No 648/2012, (EU) No 600/2014, (EU) No 909/2014 and (EU) 2016/1011. Official Journal of the European Union, L 333, pp. 1–79 (2022), https://eur-lex.europa.eu/eli/reg/2022/2554/oj/eng, published online: Dec. 2022; Into force: Jan. 2023; Last access: March 2026
14. Feng, E., Lu, X., Du, D., Yang, B., Jiang, X., Xia, Y., Zang, B., Chen, H.: Scalable Memory Protection in the PENGLAI Enclave. In: 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). pp. 275–294. USENIX Association (Jul 2021), https://www.usenix.org/conference/osdi21/presentation/feng
15. French Cybersecurity Agency (ANSSI): Technical Position Paper on Confidential Computing, https://messervices.cyber.gouv.fr/documents-guides/anssi-technical-position-paper-coco-v1.0.pdf, published online: Oct. 2025; Last access: March 2026
16. Goltzsche, D., Nieke, M., Knauth, T., Kapitza, R.: Acctee: A webassembly-based two-way sandbox for trusted resource accounting. In: 20th International Middleware Conference. pp. 123–135 (2019)
17. Hunt, G.D.H., Pai, R., Le, M.V., Jamjoom, H., Bhattiprolu, S., Boivie, R., Dufour, L., Frey, B., Kapur, M., Goldman, K.A., Grimm, R., Janakirman, J., Ludden, J.M., Mackerras, P., May, C., Palmer, E.R., Rao, B.B., Roy, L., Starke, W.A., Stuecheli, J., Valdez, E., Voigt, W.: Confidential computing for OpenPOWER. In: 16th European Conference on Computer Systems. pp. 294–310. Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3447786.3456243
18. Intel: Intel Software Guard Extensions (Intel SGX), https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html, published: 2015; Last access: March 2026
19. Intel: Intel Trust Domain Extensions (Intel TDX), https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/trust-domain-extensions.html, published: 2021; Last access: March 2026
20. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: ACM SIGOPS 22nd Symposium on Operating Systems Principles. pp. 207–220. SOSP '09, Association for

Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1629575.1629596
21. LeCun, Y., Cortes, C.: The MNIST database of handwritten digits, https://www.lri.fr/~marc/Master2/MNIST_doc.pdf, published: 2005; Last access: March 2026
22. Lee, D., Kohlbrenner, D., Shinde, S., Asanovic, K., Song, D.: Keystone: An Open Framework for Architecting Trusted Execution Environments. In: 15th European Conference on Computer Systems. EuroSys'20 (2020)
23. Ménétrey, J., Pasin, M., Felber, P., Schiavoni, V.: Twine: An embedded trusted runtime for webassembly. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE). pp. 205–216. IEEE (2021)
24. Ménétrey, J., Pasin, M., Felber, P., Schiavoni, V.: WaTZ: A trusted WebAssembly runtime environment with remote attestation for TrustZone. In: 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS). pp. 1177–1189. IEEE (2022)
25. Ménétrey, J., Pasin, M., Felber, P., Schiavoni, V., Mazzeo, G., Hollum, A., Vaydia, D.: A Comprehensive Trusted Runtime for WebAssembly with Intel SGX. IEEE Transactions on Dependable and Secure Computing (2023)
26. Michaud, Q., Ramezanian, S., Ayed, D., Levillain, O., Garcia-Alfaro, J.: Abstraction of Trusted Execution Environments as the Missing Layer for Broad Confidential Computing Adoption: A Systematization of Knowledge (2025), https://doi.org/10.48550/arXiv.2512.22090
27. Qiang, W., Dong, Z., Jin, H.: 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. pp. 451–470. Springer (2018)
28. Rossberg, A.: Webassembly specification, https://webassembly.github.io/spec/core/, published: 2025; Last access: March 2026
29. Sahita, R., Shanbhogue, V., Bresticker, A., Khare, A., Patra, A., Ortiz, S., Reid, D., Kanwal, R.: CoVE: Towards confidential computing on RISC-V platforms. In: 20th ACM International Conference on Computing Frontiers. pp. 315–321 (2023)
30. The CONFIDENTIAL6G project: Confidential WebAssembly runtime for Keystone enclave (2025), https://github.com/vector-sdk/wasm-demo, published online: 2025; Last access: March 2026
31. Tsai, C.C., Porter, D.E., Vij, M.: Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In: 2017 USENIX Annual Technical Conference (USENIX ATC 17). pp. 645–658. USENIX Association, Santa Clara, CA (2017), https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai
32. Valdez, E., Ahmed, S., Gu, Z., de Dinechin, C., Cheng, P.C., Jamjoom, H.: Crossing Shifted Moats: Replacing Old Bridges with New Tunnels to Confidential Containers. In: ACM SIGSAC Conference on Computer and Communications Security. pp. 1390–1404. CCS'24, Association for Computing Machinery, New York, NY, USA (2024), https://doi.org/10.1145/3658644.3670352
33. Wang, W.: How Far We've Come – A Characterization Study of Standalone WebAssembly Runtimes. In: 2022 IEEE International Symposium on Workload Characterization (IISWC). pp. 228–241 (Nov 2022). https://doi.org/10.1109/IISWC55918.2022.00028
34. Zhang, Y., Liu, M., Wang, H., Ma, Y., Huang, G., Liu, X.: Research on webassembly runtimes: A survey. ACM Trans. Softw. Eng. Methodol. **34**(8) (Oct 2025). https://doi.org/10.1145/3714465