TRUST WITHOUT EXPOSURE: VERIFIABLE OBSERVABILITY WITH CAPABILITY-NATIVE WEBASSEMBLY AT THE EDGE

Bala Subramanyan

Verifoxx, London, UK

ABSTRACT

*In modern data ecosystems, where edge autonomy, privacy, and verifiability are essential, enabling trustworthy observability without compromising data control remains a significant challenge. This paper presents **cWAMR**, a capability-native WebAssembly runtime adapted for the **CHERI** (Capability Hardware Enhanced RISC Instructions) architecture, enabling fine-grained, hardware-enforced compartmentalization of untrusted code.*

*We demonstrate how **cWAMR** enables the construction of **Verifiable Observability Pipelines (VoP)**—modular, staged execution flows deployed across edge environments. Each pipeline stage is implemented as an isolated WebAssembly module running in a CHERI-sealed cWAMR compartment, with capability-based delegation enforcing tamper-evident data flow and memory safety without shared linear memory or enclave-based trust models.*

Deployed and validated on the Arm Morello platform under the UK DSbD initiative, cWAMR supports both interpreted and ahead-of-time WebAssembly execution, integrated with CHERI-aware system interfaces (cWASI). The result is a lightweight, privacy-aligned foundation for building observable, compliant edge pipelines—enabling cryptographically anchored provenance and lifecycle assurance without cloud dependency or centralised attestation infrastructure.

KEYWORDS

WebAssembly, CHERI, Capability-Based Security, Verifiable Observability, Memory Safety, Data Provenance, Privacy Preserving Pipelines, Data as a Product (DaaP)

1. INTRODUCTION

1.1 MOTIVATION

As digital ecosystems increasingly embrace the paradigm of “Data as a Product” [39], the need for verifiable observability pipelines [38][40] at the edge has become urgent. Institutions and data providers require the ability to process data locally - close to the source, without ceding control, exposing raw datasets, or relying on heavy cloud infrastructure and centralized attestation schemes [4]. This demand is particularly pronounced in regulated domains, where privacy, auditability, and governance requirements conflict with monolithic, cloud-first architectures.

WebAssembly (WASM) [12] has emerged as a versatile and language-neutral runtime format, now widely deployed across browsers, cloud-native microservices, edge gateways, and embedded devices. By abstracting execution into a platform-independent bytecode model with a linear memory interface, WASM offers a portable, lightweight, and sandboxed mechanism to run untrusted workloads [30].

However, WASM's sandboxing is enforced entirely in software, through dynamic bounds checks and memory protections. These mechanisms introduce significant runtime overhead and remain vulnerable to speculative and transient execution attacks [33]. Moreover, when WASM is embedded within Trusted Execution Environments (TEEs) such as Intel SGX [11], new challenges arise: enclave transition penalties, side-channel risks, and centralized trust dependencies. These limitations make TEEs ill-suited for large-scale edge deployments or multi-tenant data pipelines [25][28].

1.2 Objectives And Contributions

To address the limitations of current WASM-based or enclave based execution in privacy-sensitive, decentralized environments—particularly the reliance on software-managed sandboxing or centralized enclave infrastructures—we propose a new architecture that combines hardware-enforced compartmentalization with verifiable, edge-resident data observability. Specifically, this paper presents:

- **cWAMR** (capability-based WebAssembly Micro Runtime) [37]:

  A CHERI-adapted execution environment derived from WAMR, designed for secure deployment of WASM modules at the edge. cWAMR integrates:
  - A sealed, capability-aware memory allocator
  - A capability-restricted system interface (cWASI)
  - Secure handling of WASM externref objects

  This runtime eliminates the need for traditional enclave-based models, offering fine-grained isolation, pointer provenance, and hardware-level spatial safety—even under speculative execution conditions. cWAMR [37] ensures that transformation logic can run securely at the data owner's premises or on constrained edge devices, without cloud dependency or attestation infrastructure.

- **Staged Verifiable Observability Pipeline (VoP)**

  Built atop **cWAMR** [37], the Verifiable Observability Pipeline (VoP) enables staged, tamper-evident data processing through capability-enforced WebAssembly compartments. Each stage operates as an isolated cWAMR runtime, with cryptographic delegation of execution outputs—eliminating shared memory and ensuring directional data flow.

  Rather than transmitting raw data, the pipeline supports the extraction of structured, non-reversible execution artifacts that reflect processing state. These artifacts can be securely dispatched to our decentralized Source of Truth (SoT) nodes for future verifications.

  While the internal logic of transformation and extraction remains **proprietary**, cWAMR enables the secure composition and isolation of such pipelines at the edge—laying the groundwork for downstream capabilities like **Proof of Provenance** (PoP) and **Predicate Disclosure Proofs** (PDPs), without requiring trusted third parties or enclave-based attestation.

- **Edge-first deployment and Validation**
-

  We validate the architecture on the Arm Morello [23] CHERI [2] platform, developed under the UK's Digital Security by Design (DSbD) [18] program. Our experiments confirm secure execution of both AoT-compiled and interpreted WASM modules under hardware-enforced capability models, demonstrating low runtime overhead and eliminating the complexity of enclave-based systems.

These contributions enable the realization of decentralized, privacy-enhancing observability pipelines—where verifiability and trust are embedded into the architecture itself, and not dependent on cloud-native infrastructure or external validators. This marks a shift toward secure, hardware-backed accountability for modern edge data systems.

1.3 Paper Organization

The remainder of this paper is structured as follows:

- **Section 2** provides background on WebAssembly runtimes, Trusted Execution Environments (TEEs), and the CHERI architecture. It also reviews related efforts in secure runtime isolation and privacy-preserving computation.

- **Section 3** presents the design of cWAMR, detailing how CHERI [1][2] capabilities are integrated into the WebAssembly Micro Runtime (WAMR) [13], including memory allocation, system interface enforcement (cWASI) [37], and secure reference handling.

- **Section 4** outlines the implementation of the Verifiable Observability Pipeline (VoP) using cWAMR's compartmentalized runtime, staged processing and secure delegation.

- **Section 5** validates cWAMR and VoP on the Arm Morello [23] platform, demonstrating correct execution, memory safety, and compatibility with Ahead-of-Time (AoT) and interpreted WASM modules under CHERI enforcement.

- **Section 6** concludes the paper, summarizing key contributions and future directions in capability-based, privacy-aligned data execution pipelines.

2. Background And Related Work

2.1 WebAssembly And Language Agnostic Runtimes

WebAssembly (WASM) [6][19] is a compact, portable bytecode format designed to serve as a compilation target for high-level languages such as C, C++, and Rust. Unlike conventional virtual machines (e.g., JVM, CLR) that rely on rich object models and managed memory, WASM adopts a minimalist execution stack and a linear memory model—representing the module's heap as a contiguous, byte-addressable array. Indexing is performed using raw integer offsets, decoupling execution semantics from any specific hardware architecture.

WASM's strength lies in its cross-platform portability and fast startup characteristics. The ecosystem is further enhanced by the WebAssembly System Interface (WASI) [14], which standardizes host access to system resources and networking interfaces, allowing WASM to operate as a general-purpose runtime outside the browser.

However, WASM’s default isolation model is implemented entirely in software. Spatial safety is enforced via dynamic bounds checks and host memory protections (e.g., mprotect), [33] which introduce performance overheads and remain susceptible to attacks exploiting speculative execution leveraging the characteristics of the underlying target host CPUs or side channels. Studies like "Leaps and Bounds" [33] report overheads of up to 650% under certain runtimes and workloads, especially when sandboxing and fine-grained checks are enforced via user-space mechanisms.

In the context of sensitive edge workloads and privacy-critical data, these limitations constrain the applicability of standard WASM runtimes. Without hardware-backed compartmentalization or pointer provenance, runtimes must rely on cryptographic wrappers (e.g., TEEs) or trust the host OS—both of which contradict the principle of minimal trust surfaces in decentralized or regulated environments [32].

This motivates a shift toward integrating WebAssembly runtimes with architectural enforcement of memory and security boundaries. In this work, we extend the WebAssembly Micro Runtime (WAMR) [13] with CHERI capabilities, enabling a hardware-backed execution layer that supports secure, efficient, and privacy-respecting observability pipelines on the edge.

2.2 TRUSTED EXECUTION ENVIRONMENTS (TEE)

Trusted Execution Environments (TEEs) provide a hardware-supported isolation zone for code and data, shielding them from the host operating system and other privileged software. Examples include Intel SGX [11], AMD SEV [17], and AWS Nitro Enclaves [16]. These environments create a trust boundary between secure workloads and potentially compromised infrastructure.

Intel SGX partitions memory into a dedicated Enclave Page Cache (EPC), encrypting memory outside the CPU and supporting remote attestation for workload verification [32]. However, SGX imposes notable limitations:

- **Transition Overheads**: Calls to and from the enclave (OCALLs/ECALLs) incur heavy marshalling and protection costs.
- **Memory Constraints**: EPC sizes are statically limited (e.g., 128–256MB), causing paging penalties when exceeded.
- **Microarchitectural Leaks**: SGX enclaves remain vulnerable to speculative execution and side-channel attacks (e.g., Spectre [7], LVI [26]).

AMD SEV [17] encrypts memory at the virtual machine level, protecting entire guest OSes but lacking intra-application or per-module compartmentalization. AWS Nitro Enclaves provide isolated execution environments but require VM-like lifecycle management and coarse-grained memory control.

In addition to these performance and scalability drawbacks, TEEs often require vendor-specific provisioning, attestation protocols, and opaque configuration, making them cumbersome for decentralized deployments or data owner-controlled environments. TEEs also do not address fine-grained control within a process—memory safety still depends on software discipline.
In this paper, we circumvent these constraints by replacing traditional enclave-based execution with capability-based security using CHERI, enabling lightweight but strong hardware isolation for WASM modules without the runtime and management burden of TEEs.

2.3 CHERI: ARCHITECTURAL CAPABILITY ENFORCEMENT

CHERI (Capability Hardware Enhanced RISC Instructions) [1][2] fundamentally rethinks memory safety by embedding protection directly into processor instructions and registers. Instead of untyped pointers, CHERI employs capabilities—128-bit enriched references that tightly couple memory addresses with metadata describing what can be accessed and how.

Each capability contains:

- A **64-bit virtual address**, indicating the base location.
- **Compressed bounds**, specifying the lower and upper memory limits.
- **Permission bits**, detailing allowed operations (e.g., load, store, execute).
- A hidden **validity tag**, atomically tracked in hardware, which ensures the capability's authenticity.

These attributes are enforced by the CPU on every memory access or control transfer:

- **Spatial safety:** Memory loads, stores, and jumps outside the authorized bounds automatically fault, blocking buffer overflows.
- **Permission safety:** Even within valid bounds, operations must match the capability's permissions, preventing writes to read-only segments or unauthorized instruction fetches.
- **Provenance and integrity:** Capabilities can only be derived (through explicit instructions like bounds narrowing or sealing) from other valid capabilities. This directly enforces temporal safety by invalidating stale references.

CHERI introduces sealed capabilities, which are cryptographically guarded by the hardware so they cannot be dereferenced or modified until explicitly unsealed. This allows creating software compartments (e.g., isolating libraries or WASM modules) where cross-compartment interaction must occur through explicitly granted capabilities, enforcing strict least-privilege boundaries within a single address space.

Furthermore, CHERI's guarantees persist even under speculative execution, where typical software checks fail. Formal models [34] have verified that CHERI enforces security invariants in presence of speculative attacks, elevating it as a trustworthy alternative to TEEs for secure execution environments.

In this work, we leverage CHERI to harden WebAssembly runtimes against memory corruption and isolation failures. By modifying WAMR to operate on capabilities and execute inside sealed compartments, we build **cWAMR [37]**, the first WASM runtime with CHERI-native memory protection, suitable for privacy-critical edge deployments.

2.4 RELATED WORK

WebAssembly runtimes and bounds enforcement

A wide body of work has explored the performance and security characteristics of WebAssembly runtimes. Early investigations like *Jangda et al.* [35] highlighted the cost of dynamic safety checks in WASM, showing that bounds checking for linear memory and indirect call tables can introduce overheads ranging from 10% to over 200% on real workloads. More recent empirical studies, such as *Leaps and Bounds* [33], benchmarked multiple WASM runtimes across x86, Arm, and RISC-V architectures, isolating the substantial cost of *mprotect*()-based memory protections and *userfaultfd* schemes typically employed by runtimes like V8 and Wasmtime [3] to implement sandboxing. These studies confirm that WASM remains sensitive to leaks and incurs notable performance tradeoffs under memory protection.

TEEs and double sandboxing of WASM

To mitigate risks in untrusted platforms, multiple efforts have embedded WASM runtimes inside Trusted Execution Environments (TEEs), layering a software sandbox within a hardware enclave to achieve "double sandboxing." For instance, *TWINE* [9] runs unmodified WASM inside Intel SGX enclaves, relying on SGX's encrypted memory and attestation to protect against compromised hosts. Similarly, *AccTEE* [10] executes WASM workloads inside AMD SEV virtual machines, leveraging full-VM memory encryption.

While these approaches raise the security bar, they inherit TEE-specific limitations: SGX suffers from EPC paging bottlenecks and costly OCALL/ECALL transitions, while SEV lacks intra-application compartmentalization [25][32]. Both remain exposed to speculative execution attacks on shared microarchitectural state, as demonstrated by Spectre [7], Meltdown [8], and LVI [26] variants. Moreover, TEEs still follow conventional ISA rules without intrinsic bounds or provenance enforcement.

Capability hardware for fine-grained memory safety

Outside TEEs, CHERI (Capability Hardware Enhanced RISC Instructions) represents a fundamentally different approach by embedding unforgeable capabilities with bounds, permissions, and validity tags directly into the CPU pipeline [1][2]. This enables hardware-enforced spatial safety, provenance-based temporal safety, and sealed capabilities for software-defined compartments, all without relying on encrypted memory or heavyweight enclave mechanisms.

To our knowledge, our work is the first to adapt a portable language-neutral WebAssembly runtime (WAMR) to execute directly within CHERI compartments, enforcing capability bounds and pointer integrity at the hardware level.

3. cWAMR Architecture

3.1 Overview Of The CHERI-Enhanced WebAssembly Runtime

The **Capability-Aware WebAssembly Runtime (cWAMR) [24]** is a CHERI-augmented variant of the WebAssembly Micro Runtime (WAMR) [13], designed to provide hardware-enforced memory safety, secure module isolation, and native execution compatibility for unmodified WebAssembly binaries.

Unlike traditional WASM runtimes, which rely on software-based sandboxing within a flat linear memory model, cWAMR [24] leverages **CHERI (Capability Hardware Enhanced RISC Instructions) [1][2]** to enforce fine-grained, hardware-level memory protection. CHERI replaces untyped pointers with capability-enforced references that include bounds, permissions, and integrity constraints, eliminating common vulnerabilities such as buffer overflows, use-after-free errors, and speculative execution attacks.

A central challenge in integrating CHERI with WebAssembly is the mismatch between WASM's linear memory abstraction and CHERI's capability-based memory model. Additionally, to support native interoperability and compatibility with the WebAssembly System Interface (WASI) [14], cWAMR [24] must bridge conventional pointer-based host interfaces with CHERI's strict capability semantics.

To address these challenges, cWAMR [24] introduces several architectural enhancements:

- A **capability-aware WASI layer (cWASI) [24]** that mediates all system calls through CHERI-sealed references, ensuring per-compartment access control.
- Secure handling of externref objects, enabling memory-safe interaction between WebAssembly modules and host-native functions.
- Support for both hybrid and purecap execution modes [2], allowing developers to incrementally adopt CHERI's full security model without sacrificing compatibility.

Through these mechanisms, cWAMR [24] achieves a novel form of **double sandboxing**, where WebAssembly's software-level isolation is nested within CHERI's hardware-enforced compartments—eliminating the need for cryptographic attestation mechanisms typically required in Trusted Execution Environments (TEEs).

3.2 Execution Models: Hybrid And PureCap

WebAssembly is traditionally designed to operate with a linear memory model, where memory is accessed through 32-bit or 64-bit untyped integer offsets. While this model supports platform-independence and predictable sandboxing, it provides limited protection against low-level memory manipulation and pointer-based vulnerabilities.

By contrast, CHERI [2] replaces raw pointers with capability-enforced references. Each reference includes metadata encoding valid memory bounds, access permissions, and

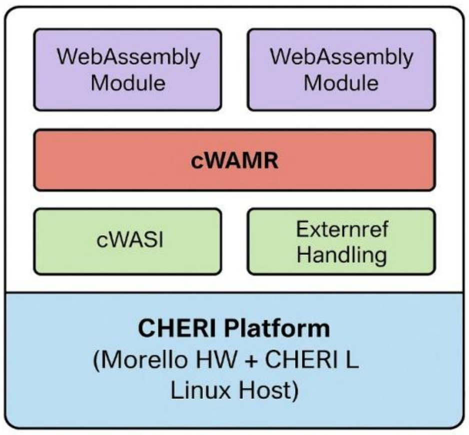

Figure 1. Layered Architecture of cWAMR on the CHERI Platform

provenance, enabling hardware-level enforcement of memory safety and compartmentalization. To accommodate diverse deployment environments and support progressive adoption, cWAMR [24] supports two execution models:

Hybrid Mode (Partial CHERI Enforcement):

- WebAssembly modules operate largely within the traditional linear memory abstraction.
- Selected memory regions and system interactions are protected using CHERI capabilities.
- This model preserves compatibility with legacy WASI applications and standard toolchains.
- CHERI-based enforcement can be selectively applied to high-risk operations, such as system calls and shared memory access.

Purecap Mode (Full Capability Enforcement):

- All memory interactions, including function arguments, return values, and heap allocations, are mediated through CHERI capabilities.
- WebAssembly memory is fully compartmentalized, and pointer manipulation outside defined bounds is hardware-trapped.
- Purecap execution provides complete memory safety and isolation, but requires CHERI-aware toolchain support (e.g., CHERI-LLVM [22]).

cWAMR's dual-mode design allows developers and platform architects to incrementally transition from traditional sandboxing models to fully hardware-enforced execution. This flexibility is critical for real-world adoption, enabling compatibility with existing WebAssembly ecosystems while gradually strengthening trust boundaries through CHERI.

3.3 cWASI - Capability Aware System Interface for WebAssembly

3.3.1 WASI Limitations in CHERI Context

The WebAssembly System Interface (WASI) [14] standardizes access to essential system resources—such as file I/O, networking, clocks, and entropy—allowing sandboxed WebAssembly modules to interact with their host environments in a platform-agnostic way.

In conventional runtimes (e.g., WAMR [13], Wasmtime [3], Lucet [31]), WASI is implemented using raw pointers and integer-based file descriptors within a linear memory model. While suitable for traditional sandboxing, this design conflicts with capability-based architectures like CHERI, which embed bounds and permissions directly into memory references.

Key limitations in this context include:

- Incompatibility with CHERI pointers: WASI APIs rely on unbounded raw pointers, which CHERI explicitly prohibits.
- Assumed trust in the host: WASI presumes a trusted environment, whereas CHERI enforces per-compartment untrust and requires explicit delegation.
- High-overhead OCALLs in enclave models: In SGX-based runtimes (e.g., TWINE [9]), WASI calls cross enclave boundaries via OCALLs, introducing cryptographic costs and side-channel exposure (e.g., Spectre [7], LVI [26]).

3.3.2 cWASI - A Secure Interface Model

To address these issues, we introduce **CHERI-WASI (cWASI) [24]** —a capability-enforced WASI implementation tailored for cWAMR. Rather than re-inventing the entire WASI spec, cWASI preserves the existing syscall semantics but ensures that:

- **Pointer arguments are capabilities**: All memory-passing operations expect CHERI "***_capability"*** types with hardware-enforced validity.
- **File descriptors and handles are sealed**: cWASI binds resources (like files or sockets) to a module's compartment using fine-grained tokens instead of ambient authority.
- **System calls run in-process**: No boundary transitions or OCALLs are needed, unlike SGX. Instead, capability checks are performed *prior* to system dispatch, maintaining latency and integrity.

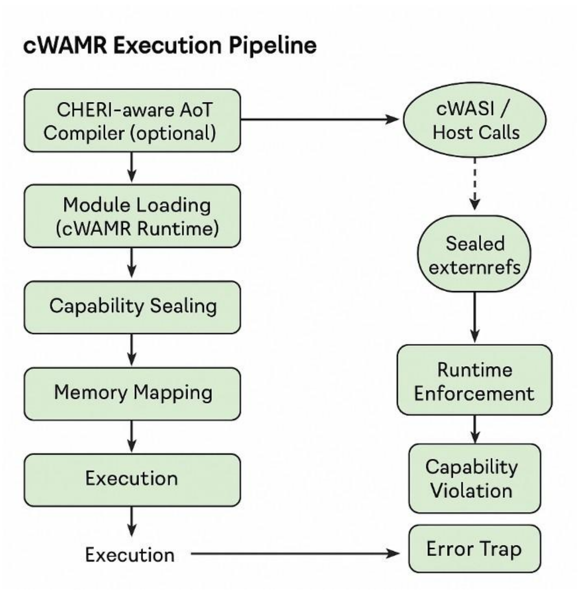

Figure 2. cWAMR Execution Pipeline Overview

3.3.3 Implementation Realities

Raw Pointer Replacement

Many WASI [14] functions (e.g., fd_read) were adapted to use "***_capability***" parameters. However, direct substitution wasn't trivial—WAMR's internals relied on linear memory assumptions. This required rewriting memory access logic in *mem_alloc* and *memcpy* paths to validate bounds via CHERI instructions instead of manual offsets.

File Access Control

Standard WASI permits open-ended path access. In cWASI, resource delegation uses sealed capability tokens, stored in per-module descriptor tables. *path_open* was rewritten to enforce token validation prior to every open syscall.

Avoiding OCALLs

By keeping WASI syscalls in-process and validating all capability arguments via CHERI intrinsics, cWASI removed the need for TEE-like attestation or memory copying across enclave edges. This significantly reduced syscall latency.

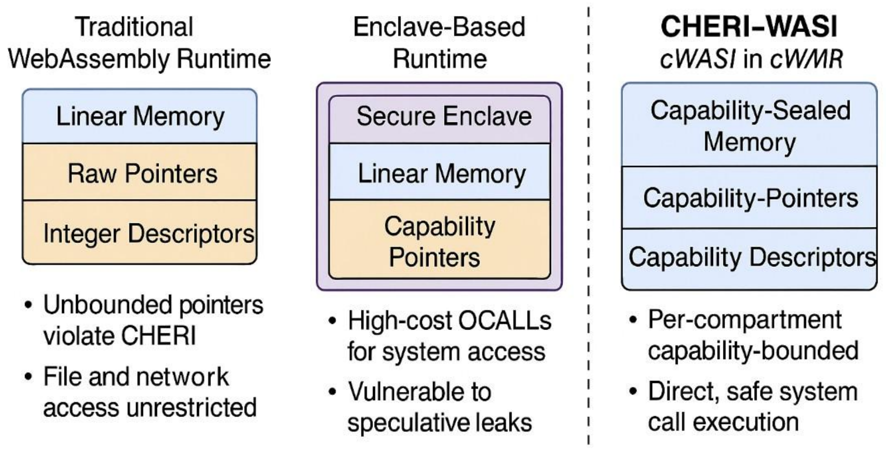
Traditional WebAssembly Runtime
Linear Memory
Raw Pointers
Integer Descriptors
• Unbounded pointers violate CHERI
• File and network access unrestricted
Enclave-Based Runtime
Secure Enclave
Linear Memory
Capability Pointers
• High-cost OCALLs for system access
• Vulnerable to speculative leaks
CHERI-WASI
cWASI in cWMR
Capability-Sealed Memory
Capability-Pointers
Capability Descriptors
• Per-compartment capability-bounded
• Direct, safe system call execution

Figure 3. Comparative Models of WASI Integration in Traditional, Enclave-Based, and CHERI-Based WebAssembly Runtimes

This design enables cWAMR to support secure, low-overhead system interactions while preserving compatibility with standard WebAssembly tooling. As a result, cWAMR stands apart as the first runtime to offer capability-native system calls, delivering a scalable and hardware-enforced foundation for secure execution beyond the limitations of TEE-based models.

3.4 Externref Handling

3.4.1 Problem: Unsafe Native Bridging

The externref [29][30] construct in WebAssembly was designed for flexibility—allowing modules to hold opaque references to host-managed objects. Unfortunately, most runtimes implemented these using global raw pointer tables. This model breaks on CHERI for two reasons:

1. **It allows type-unsafe access**: An externref [30] could be used across modules with incompatible layouts.
2. **It lacks memory provenance**: Once allocated, externrefs could outlive their owners, risking stale or hijacked pointers.

3.4.2 Our Solution: Capability Wrapped Object References

In **cWAMR [24]**, we redesigned externref handling to behave more like *capability-based handles* instead of raw table indices:

- **Reference Table Rewritten**: We replaced WAMR's static externref table with a dynamic slab allocator that stores CHERI-sealed capabilities per module context.
- **Lifetime Linking**: Capabilities were bound to the lifecycle of their owning module. Once a module is deallocated, all associated capabilities are invalidated using CHERI's provenance model—removing the need for manual reference counting or finalization hooks.
- **Cross-Compartment Passing**: To support externref usage across modules, cWAMR uses sealed delegation. Only capabilities explicitly passed through host exports (e.g., via **wasm_export_function**) are valid in downstream modules.

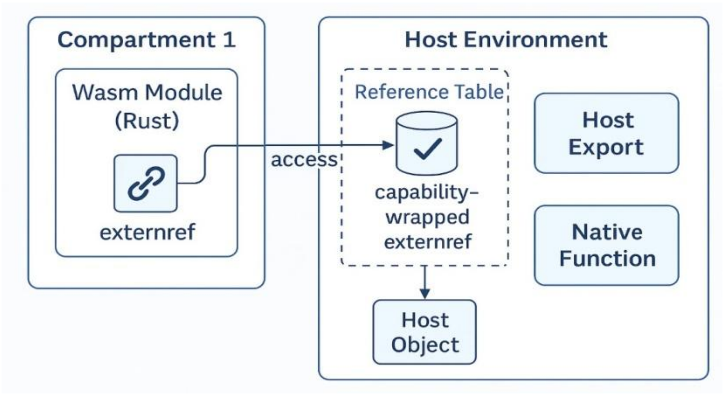

Figure 4. Secure wasm interop using capability-wrapped externrefs

3.4.3 Native Execution With Secure References

cWAMR allows WASM modules to call native functions using externrefs [29] that are now CHERI capabilities. Here's how we enforce safety without performance compromise:

- Native functions only dereference memory through capability-validated pointers.
- If a module tries to pass a forged or expired reference, the CPU triggers a hardware fault—pre-empting attack attempts.
- Because all operations are in-process, there's no cryptographic attestation step (as in SGX), and calls occur at near-native speed.

By enforcing memory provenance, compartment scoping, and revocation at the hardware level, cWAMR's externref system brings deterministic safety to a historically error-prone interaction layer. It ensures that native interoperability no longer undermines the isolation promised by WebAssembly.

3.5 Fine Grained Compartmentalization in cWamr

In conventional TEE-backed WebAssembly runtimes like TWINE [9] (SGX) or Enarx [15] (SEV/TDX), all WebAssembly modules typically execute within a monolithic secure enclave. While this offers memory confidentiality against untrusted OS or hypervisors, it fails to enforce intra-enclave isolation between multiple modules. Modules share the same virtual address space, creating risks of:

- Intra-tenant data leakage
- Unbounded pointer misuse
- Privilege escalation across modules
- Exposure to speculative attacks (e.g., Spectre [7], LVI [26])

cWAMR's Capability-Enforced Isolation

Unlike enclave-based models, cWAMR uses CHERI's architectural primitives [1] to allocate and seal individual capability domains per module. Each WebAssembly module is executed inside its own capability-constrained compartment, configured at runtime by a capability manager. This enforces hardware-enforced compartment boundaries that cannot be bypassed in software, eliminating reliance on encrypted paging or cryptographic attestation.

Key Design Features

- **Compartment-Scoped Memory Maps**
  Each WASM module's heap, operand stack, and frame stack are sealed as independent capability regions. Memory accesses outside a compartment's bounds trigger hardware-enforced CHERI faults, protecting against out-of-bounds memory manipulation or pointer aliasing.
- **Cross-Module Delegation via Sealed Capabilities**
  Unlike enclave models that rely on RPC [9] marshalling or OCALL [11] patterns, cWAMR implements capability transfer through explicitly delegated sealed objects. Capabilities passed across modules are derived from narrowed parent objects, limiting authority propagation.
- **No Shared Linear Memory by Default**
  WebAssembly's default linear memory model is kept private to each compartment unless explicitly shared via CHERI-sealed objects (e.g., shared memory pools or cross-compartment tables), avoiding unintentional leakage.
- **Secure Code Re-Entrancy and Switching**
  Module calls, recursion, and system interactions use a capability-aware call stack switcher that preserves isolation and avoids capability corruption during context transitions.

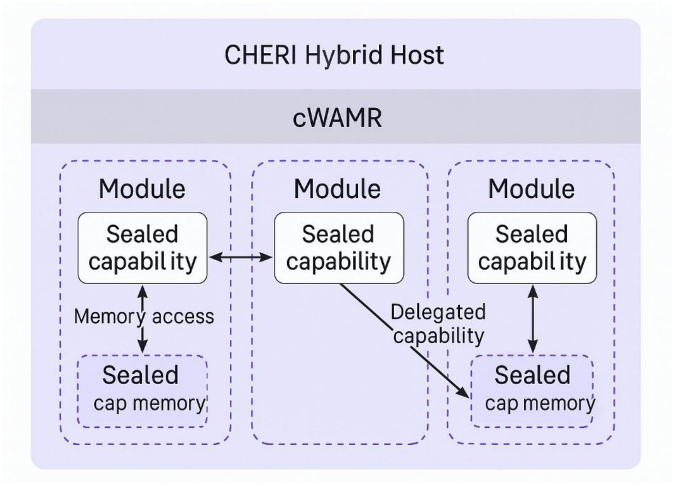

Figure 5. Per-Module Isolation and Capability Delegation in cWAMR Implementation Considerations

- The CHERI-LLVM–compiled [22] AoT modules embed per-function sealed entry points, allowing controlled invocation by a capability dispatcher.
- Runtime validation enforces that entry capabilities are not reused or passed backward from callee to caller, maintaining directional integrity of access.
- Interactions via cWASI or externref are scoped to the current compartment's derived capabilities, ensuring that host-facing calls cannot be misused by other tenants.

Unlike enclave models constrained by cryptographic boundaries and shared secure memory, cWAMR ensures that each module is an isolated, non-overlapping security domain enforced by CHERI's hardware. This architecture eliminates privilege flattening, prevents capability reuse, and supports scalable multi-tenant isolation without performance-heavy TEE constructs.

Table 1. Runtime Isolation Features: cWAMR vs. Enclave based approaches

Feature	cWAMR (CHERI)	SGX-based WebAssembly
Memory Isolation	Per-module, capability enforced	Enclave-wide; shared threads
System Call Semantics	In-process, sealed via cWASI; capability bound	OCALL-based; boundary transitions
Externref handling	Sealed delegated references	Raw pointers; manual validation
Module Switching	Hardware-regulated compartment transitions	Software switches within single enclave
Inter-Module Communication	Delegated capabilities only	Shared memory or manual software guards
Speculative Attack Surface	Narrowed via per-compartment CHERI sealing	Shared enclave state susceptible to leakage

3.6 Security Model

The security design of **cWAMR** is rooted in CHERI's hardware-backed capability system, which enforces spatial memory integrity, provenance validity, and explicit compartment boundaries at the instruction level—eliminating dependence on cryptographic attestation, encrypted paging, or external marshalling commonly required in TEE-based designs.

Hardware-Enforced Memory Integrity and Capability Provenance

In cWAMR, all internal runtime structures—including stack frames, linear memories, and external references—are represented as CHERI capabilities. Each capability tightly couples:

- bounds that constrain valid address ranges,
- fine-grained permissions (read, write, execute, seal),
- and provenance metadata that tracks derivation chains.

This ensures that memory accesses cannot exceed their authorized object boundaries, fabricated pointers are invalidated by tag checks, and stale references after deallocation cannot regain privileges—directly preventing classes of vulnerabilities such as buffer overflows, use-after-free, and pointer aliasing attacks.

Speculative and Out-of-Order Safety

Unlike traditional sandboxed runtimes or TEEs that remain vulnerable to transient execution attacks due to speculative misuse of stale or forged pointers, cWAMR leverages CHERI's architectural guarantees. Specifically:

- The capability check pipeline ensures that speculative loads cannot dereference invalid or out-of-bounds pointers, in alignment with CHERI's proposed Capability Speculation Contracts (CSC).
- This prevents speculative memory leaks through bounds or permissions violations, maintaining the invariant that no memory access can occur without architectural authorization.

By executing within a sealed, capability-constrained compartment, cWAMR inherently minimizes side-channel exposure surfaces tied to speculative control flow or indirect jumps.

Explicit Capability-Scoped Multi-Tenancy

cWAMR [37] implements a strict multi-compartment model, where each WebAssembly module runs inside its own CHERI compartment with dedicated sealed capabilities. There is no implicit sharing of memory or resources:

- Inter-module communication, shared buffers, or host API accesses require explicit capability delegation.
- This zero-trust design sharply contrasts with enclave-based TEEs, which typically assume a single large trusted memory region for all enclave code, raising risks of internal privilege escalation.

Eliminated Trusted Host Dependence

By enforcing strict memory boundaries and execution provenance at the hardware level, cWAMR [37] creates trustworthy isolation domains from which cryptographic transformation fingerprints can be reliably extracted. This provides the basis for later verifiability steps—such as proving the origin and context of data processing (Proof of Provenance)—without requiring enclave attestation or trusted intermediaries.

Traditional TEEs or sandboxed runtimes often rely on external marshalled OCALLs (e.g., for file I/O or cryptographic operations) that expose privileged host interfaces to untrusted guest data. In contrast, cWAMR's integration of cWASI [37] ensures that all system interactions are mediated through capability-qualified interfaces, with:

- no unbounded raw pointers crossing runtime boundaries,
- no dependence on privileged host code outside the CHERI trust perimeter.

This substantially reduces the trusted computing base (TCB) and simplifies formal reasoning about security, aligning with CHERI's goals of minimal, well-defined hardware-enforced software compartments.

cWAMR's architectural design aligns closely with privacy-enhancing technologies (PETs) [20], offering a low-trust, formally bounded environment ideal for edge-based data processing pipelines. It supports cryptographic accountability without disclosing source data or depending on centralized control.

4. Implementation And System Integration

The cWAMR [24] runtime is derived from the WebAssembly Micro Runtime (WAMR) [13] and has been deeply refactored to align with CHERI's capability system. The implementation focuses on replacing unsafe linear memory operations with CHERI-enforced references while preserving compatibility with unmodified WebAssembly binaries.

4.1 CHERI Adaptations in WAMR

To make the WAMR runtime CHERI-compliant, several subsystems were overhauled:

- **Memory Allocation**: The mem_alloc, heap_malloc, and runtime linear memory initialization functions were updated to use CHERI-safe memory via _builtin_cheri_bounds_set and related intrinsics. This ensures that memory blocks returned from the allocator are bounds-restricted and provenance-tracked.
- **ExecEnv Refactoring**: The exec_env context was modified to hold sealed capabilities for the stack and runtime frame pointers. This prevents frames or modules from forging or traversing invalid stack memory during WASM invocation.
- **CHERI-Safe Host Intrinsics**: Built-in hostcalls (e.g., memcpy, strcpy, and indirect function tables) were patched to validate capabilities before dereferencing. Unsafe constructs like mem_access_addr = base + offset were eliminated and replaced with capability-aware access patterns.
- **System Call Interface**: WASI call sites were entirely rewritten in native_wasi_api.c to replace raw pointer arguments with *_capability*-qualified parameters. The internal function tables, WASM-native to host mappings, and argument unwrapping logic were modified to decode and enforce CHERI metadata.

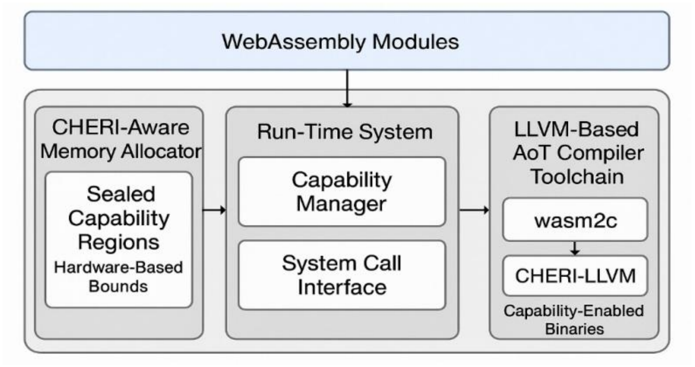
WebAssembly Modules
CHERI-Aware Memory Allocator
Sealed Capability Regions
Hardware-Based Bounds
Run-Time System
Capability Manager
System Call Interface
LLVM-Based AoT Compiler Toolchain
wasm2c
CHERI-LLVM
Capability-Enabled Binaries

Figure 6. Implementation

4.2 Toolchain And Build Integration

cWAMR modules are compiled using a multi-stage CHERI-native pipeline to ensure end-to-end capability safety:

- **AoT Path (wasm2c → CHERI-LLVM)**: WebAssembly modules are converted to ANSI C using wasm2c. The resulting source is then compiled using CHERI-Clang (cheri-clang) [22] targeting hybrid or purecap mode (-mabi=purecap)[23]. This preserves the WASM logic while emitting capability-enforced ELF modules.

- **Object Wrapping**: Each compiled module is statically linked with a minimal CHERI-safe runtime shim, which initializes the exec_env, populates sealed memory regions, and registers capability-secure host imports. The linker script ensures that function pointers and tables reside in compartmentalized address spaces.
- **Linkage with cWAMR Core**: The CHERI-safe ELF [22] objects are integrated with the modified cWAMR runtime, including patched app_manager, runtime_memory, and native_symbol modules to handle sealed references, validated host imports, and cross-module delegation.
- **Target Validation**: The final binary is deployable on CHERI-enabled QEMU [23] and Morello platforms. Execution is validated using capability trap monitoring (cheri_ccheck_fail), runtime permission tracing, and AoT validation against native WAMR output to ensure semantic consistency.

This toolchain enables capability-native WebAssembly execution with hardware-enforced security guarantees. This toolchain forms the secure execution foundation over which verifiable observability pipelines can extract data transformations with minimal runtime overhead.

4.3 VERIFIABLE OBSERVABILITY PIPELINE

The Verifiable Observability Pipeline (VoP) is a staged, compartmentalized framework built atop the cWAMR runtime [24], designed to support cryptographically verifiable transformations at the edge—without disclosing raw data, fingerprinting logic, or relying on centralized trust anchors.

At its core, VoP leverages **cWAMR's CHERI-enforced compartments [24]** to isolate and orchestrate WebAssembly (WASM) modules, enabling traceable, bounded, and non-reversible execution. Each stage operates within an independent memory-safe runtime compartment, communicating only via **explicit capability delegation**, thus forming a privacy-preserving, tamper-evident execution chain.

> **Note:** While the architectural flow is presented here in full, the internal logic of fingerprinting, transformation, and sealing modules remains proprietary. These operations are securely deployed within compartments, and are not disclosed in this paper.

Pipeline Staging and Execution Flow

The Verifiable Observability Pipeline operates as a sequence of capability-isolated cWAMR stages, with no shared memory or raw pointer exchange between them. Instead, intermediate outputs are passed via sealed, unforgeable capabilities, preserving compartmental boundaries and preventing reuse or leakage. Below is a high-level view of the staged execution:

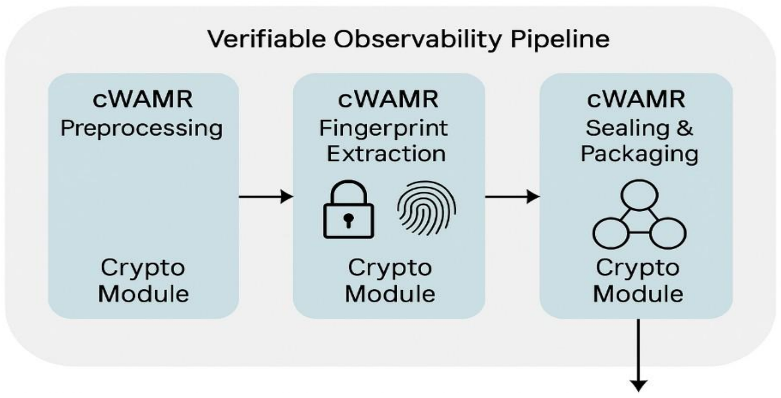

Figure 7. Sample Verifiable Observability Pipeline with cWAMR Isolation Stage 1: Preprocessing (cWAMR Compartment)

Raw input data is prepared by WASM modules provisioned into sealed cWAMR compartments. This includes normalization, schema validation, or anomaly detection—executed entirely within hardware-bounded memory.

- Output is passed forward via sealed capability handles only.

Stage 2: Fingerprint Extraction (cWAMR Compartment)

Preprocessed data is consumed by a separate, isolated fingerprinting module. Within this compartment, data is transformed into non-reversible cryptographic representations.

- Delegated references, not raw data, are used to access prior outputs.

Stage 3: Sealing & Compartmental Packaging (cWAMR Compartment)

A downstream compartment derives session-specific secrets and seals the fingerprint outputs.

- Sealing is contextual and traceable—linked to prior-stage metadata and runtime attestations.
- Payloads are made non-linkable and scoped strictly to the compartment's authority.
- At no point does unsealed data or raw computation leave any compartment boundary.
- This model eliminates the need for enclave-based marshalling or shared state across the pipeline.

Security and Deployment Guarantees of VoP

The Verifiable Observability Pipeline (VoP) enforces strong security boundaries and trustable execution across all stages:

- **Isolated Execution**: Each WASM module runs within its own CHERI-enforced cWAMR compartment, with hardware-level enforcement of bounds, memory provenance, and access permissions.
- **Directional, Verifiable Linking:** Data transitions between stages are strictly directional and cryptographically scoped.
- **Dynamic and Auditable Deployment**: Modules can be orchestrated, updated, or revoked without altering the trust assumptions or leaking execution state—supporting edge agility and compliance.
- **Privacy-Preserving Metadata Capture**: Execution never leaks source data or internal module logic, supporting selective disclosure and regulatory alignment.

This model, forms the foundation for **privacy-aligned, edge-resident analytics** where **data sovereignty, security, and verifiability** are preserved across stages. Built on open-source cWAMR under DSbD [18], the VoP model enables secure collaboration and trusted data workflows—without the complexity or opacity of enclave-based systems.

5. Validation

To verify cWAMR's correctness and readiness for CHERI-enforced environments, we built a comprehensive validation framework integrated into the Verifoxx cWAMR repository. Benchmarks are automated via a custom autorun script and organized for both hybrid-mode and purecap deployments on the Arm Morello SoC [23].

Benchmark Suites

Adapted to CHERI via dedicated scripts:

- **CoreMark**, **Dhrystone**, **Polybench**, and **Sightglass**—each includes a *cheri_build.sh* wrapper for hybrid and purecap compilation, ensuring seamless integration into the *CMakePresets.json* build system.

Execution Modes

Workloads are exercised in:

- **Interpreted** mode: using the CHERI-modified WAMR core.
- **Ahead-of-Time (AoT)** mode: employing wasm2c + CHERI-LLVM and sealed memory regions.

Automated Harness

- The autorun_benchmark script detects the target—Morello hybrid or purecap—launches applicable runtime, optionally builds AoT modules, executes tests, and aggregates results for analysis.

5.1 Functional Validation And Vop Execution Integrity

To validate the correctness and security guarantees of the cWAMR runtime as deployed in the

Verifiable Observability Pipeline (VoP), we implemented an integrated validation framework that tests capability enforcement, secure stage chaining, and tamper-evident execution flows on CHERI-enabled targets, including the Arm Morello SoC and CHERI-QEMU [23].

Key outcomes:

- Each stage ran inside a cWAMR-protected compartment, with successful enforcement of memory bounds, permission bits, and provenance checks.
- Inter-stage hand-offs occurred through **sealed capability delegation**, and any attempt to access unauthorized or expired references triggered CHERI faults.
- No shared linear memory was observed across compartments, confirming strict hardware isolation.
- Fingerprint outputs sealed in Stage 3 were successfully validated as tamper-evident when received by the Source of Truth, using only metadata and cryptographic tags—not raw data.
- System-level interactions were properly scoped via the cWASI layer, with no raw pointer or ambient authority leakage.

These results validate that:

- The porting of WAMR to CHERI is successful and stable.
- Enforces **hardware-rooted compartmentalization** across chained WASM stages.
- Enables **secure capability-scoped delegation** for verifiable inter-stage workflows.

5.2 Next Steps

With the current implementation, cWAMR [37] has successfully demonstrated secure and stable execution of WebAssembly modules on CHERI-enabled platforms, validating key architectural goals including memory safety, compartmentalized execution, and CHERI-compliant system interfacing. These outcomes establish a strong technical foundation for capability-aware WebAssembly runtimes.

Having established the architectural correctness and stability of cWAMR on CHERI platforms, our immediate roadmap focuses on:

- **Comprehensive benchmarking:** quantifying execution overheads introduced by CHERI capability checks versus software bounds models, under typical WASM computational and I/O-heavy loads.
- **Pipeline Automation and CI Integration**: Extend autorun frameworks to support *VoP staging*, per-session instantiation, and aggregation for multiple data owners at scale.
- **Capability-Aware Audit Trails**: Integrate provenance-preserving tracing mechanisms to track fingerprint generation, sealing, and dispatch—without exposing core logic or compromising privacy.
- **Performance Profiling Under Load**: Evaluate end-to-end latency across VoP stages, comparing cWAMR to enclave-based and software-only sandboxing models.

The benchmarking harness—along with build scripts, test presets, and AoT integrations—is being actively extended to support these goals. Progress is tracked in the Verifoxx cWAMR repository, with upcoming updates focused on scaling test suites and reducing integration friction. With this cWAMR aims to mature into a developer-friendly platform for secure, high-assurance WebAssembly applications in both research and production environments.

6. Conclusion

This paper presents **cWAMR**, a capability-native WebAssembly runtime purpose-built to operate atop **CHERI's hardware-enforced memory protection model**. By embedding bounds checking, permission semantics, and provenance enforcement into the execution layer, cWAMR enables fine-grained isolation and secure-by-default execution for untrusted WebAssembly modules—eliminating the need for enclave-based or software-only sandboxing models.

Validated across both hybrid and purecap CHERI configurations, cWAMR supports secure system interactions via **cWASI**, sealed externref handling, and per-module compartmentalization. These properties establish cWAMR as a reliable foundation for secure, scalable, and formally auditable execution environments.

Building on this foundation, we rearchitect our internal cryptographic operations into a **Verifiable Observability Pipeline (VoP)**—a privacy-preserving deployment model that orchestrates modular WASM binaries across compartmentalized cWAMR runtimes at the edge. The VoP design enables rapid, secure execution and verifiability without compromising data locality or introducing centralized trust assumptions, showcasing how cWAMR can support modern privacy-enhancing workflows.

As an open-source contribution under the **UK Digital Security by Design (DSbD)** initiative, cWAMR lays the groundwork for a new class of secure-by-construction WebAssembly runtimes. Future work will focus on performance tuning, compiler-level optimizations, and broader deployment enablement—advancing cWAMR and VoP as key enablers for high-assurance, privacy-preserving, multi-tenant computing systems.

Acknowledgements

This work was supported by **UK Research and Innovation (UKRI)** under the **Digital Security by Design (DSbD)** programme. We acknowledge **Dr. Georgios Papadakis**, Senior Innovation Lead for DSbD & Cyber Capabilities at UKRI, for his role in coordinating the programme and providing valuable guidance throughout the project lifecycle.

We also thank the broader DSbD and CHERI Morello ecosystem for the infrastructure, tooling, and documentation that enabled this research.

REFERENCES

[1] Watson, R. N. M., Neumann, P. G., Woodruff, J., Anderson, J., & Moore, S. W. (2019). Capability Hardware Enhanced RISC Instructions (CHERI):.University of Cambridge.

[2] Woodruff, J., Watson, R., Roe, M., et al. (2020). CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. IEEE Symposium on Security and Privacy (S&P).

[3] Wasmtime Runtime Documentation, Bytecode Alliance. [Online]. Available: https://docs.wasmtime.dev/

[4] R. Souza, J. Skluzacek, and R. Wilkinson, "Towards Lightweight Data Integration using Multi-workflow Provenance and Data Observability" arXiv preprint arXiv:2308.09004, 2023.

[5] Shinde, S., et al. (2017). PANOPLY: Low-TCB Linux Applications with SGX Enclaves..

[6] Bhardwaj, P., et al. (2021). Scaling Secure Computation with WebAssembly. IEEE CC.

[7] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, and W. Haas, "Spectre Attacks: Exploiting Speculative Execution," in 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2019, pp. 1–19, doi: [10.1109/SP.2019.00002].

[8] M. Lipp et al., "Meltdown: Reading Kernel Memory from User Space," in USENIX Security Symposium, 2018.

[9] J. Ménétrey, M. Pasin, P. Felber, and V. Schiavoni, "Twine: An Embedded Trusted Runtime for WebAssembly," in Proc. 2021 IEEE 37th Int. Conf. on Data Engineering (ICDE), Apr. 2021, pp. 161–172, doi: 10.1109/ICDE51399.2021.00025

[10] D. Goltzsche, M. Nieke, T. Knauth, and R. Kapitza, "AccTEE: A WebAssembly-based Two-way Sandbox for Trusted Resource Accounting," in Proc. of the 20th Int. Middleware Conf., 2019, pp. 123–135, doi: [10.1145/3361525.3361541].

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

[12] WebAssembly Community Group, "WebAssembly Core Specification," 2023. [Online]. Available: [https://webassembly.github.io/spec/core/]

[13] Bytecode Alliance, "WebAssembly Micro Runtime (WAMR)," [Online]. Available: [https://bytecodealliance.github.io/wamr.dev]

[14] WebAssembly System Interface (WASI), "Official Documentation," 2023. [Online]. Available: [https://wasi.dev]

[15] Y. Lu et al., "Enarx: Secure WebAssembly via Hardware Enclaves," Linux Foundation Confidential Computing Consortium, 2020. [Online]. Available: [https://enarx.dev]

[16] Amazon Web Services, "AWS Nitro Enclaves," 2021. [Online]. Available: [https://aws.amazon.com/ec2/nitro/nitro-enclaves/]

[17] AMD, "Secure Encrypted Virtualization (SEV) Overview," 2020. [Online]. Available: [https://www.amd.com/en/developer/sev.html]

[18] D. Clarke et al., "Digital Security by Design (DSbD): Programme Overview," UKRI, 2023. [Online]. Available: [https://www.dsbd.tech]

[19] A. Haas et al., "Bringing the Web Up to Speed with WebAssembly," in Proc. of the 38th ACM SIGPLAN Conf. on Programming Language Design and Implementation, 2017, pp. 185–200.

[20] OECD, "Emerging Privacy Enhancing Technologies: Current Regulatory and Policy Approaches,"OECD Digital Economy Papers, No. 3512, Mar. 2023.

[21] B. Özkale and S. Üsküdarlı, "A Survey and Guideline on Privacy Enhancing Technologies (PETs) for Collaborative Machine Learning," ResearchGate, 2023.

[22] LLVM Project, "CHERI-LLVM Compiler Infrastructure," 2023. [Online]. Available: [https://github.com/CTSRD-CHERI/llvm-project]

[23] ARM Ltd., "Morello Architecture Reference Manual," Version 1.0, 2022. [Online]. Available: [https://developer.arm.com/architectures/cpu-architecture/a-profile/morello]

[24] Verifoxx Ltd., "CHERI-WAMR Open Source Repository," 2023. [Online]. Available: [https://github.com/Verifoxx-LTD/verifoxx-cheri-wamr]

[25] S. van Schaik et al., "SoK: SGX.Fail: How Stuff Gets Exposed," IEEE Security & Privacy, 2022. [Online]. Available: [https://sgx.fail]

[26] LVI Authors, "LVI: Hijacking Transient Execution with Load Value Injection," 2020. [Online]. Available: [https://lviattack.eu/]

[27] N. He, S. Cao, H. Wang, Y. Guo, and X. Luo, "The Promise and Pitfalls of WebAssembly: Perspectives from the Industry," arXiv preprint arXiv:2503.21240, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2503.21240

[28] G. Perrone and S. P. Romano, "WebAssembly and Security: A Review," arXiv preprint arXiv:2407.12297, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2407.12297

[29] externref, Rust documentation. [Online]. Available: https://docs.rs/externref/latest/externref/

[30] P. P. Ray, "An Overview of WebAssembly for IoT: Background, Tools, State-of-the-Art, Challenges, and Future Directions," Future Internet, vol. 15, no. 8, p. 275, Aug. 2023. [Online]. Available: https://doi.org/10.3390/fi15080275

[31] Lucet: A WebAssembly Compiler and Runtime, Bytecode Alliance, GitHub repository. [Online].Available: https://github.com/bytecodealliance/lucet

[32] L. Chen, Z. Li, Z. Ma, Y. Li, B. Chen, and C. Zhang, "EnclaveFuzz: Finding Vulnerabilities in SGX Applications," in Proc. 2024 Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, Feb. 2024.

[33] Gabriel Sewczyk, Raul Gruber, Paul Patras, and Boris Köpf. "Leaps and Bounds: Analysing WebAssembly Bounds Checking." (OOPSLA) Vol. 6, Article 134, 2022. https://doi.org/10.1145/3563318

[34] Franz A. Fuchs, Jonathan Woodruff, Peter Rugg, Alexandre Joannou, Jessica Clarke, John Baldwin, Brooks Davis, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. "Safe Speculation for CHERI."

[35] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code," in Proc. 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2018, pp. 107–126.

[36] M. El-Hindi, T. Ziegler, M. Heinrich, A. Lutsch, Z. Zhao, and C. Binnig, "Benchmarking the Second Generation of Intel SGX Hardware," In Proc. DaMoN'22: Data Management on New Hardware, Philadelphia, PA, USA, June 2022. DOI: https://doi.org/10.1145/3533737.3535098

[37] Bala Subramanyan, "cWAMR: Reimagining a capability based WebAssembly Runtime via CHERI-based compartmentalization," in Proceedings of the 14th International Conference on Information Processing, Data Computing and Analytics (IPDCA), London, UK, May 2025. [Online]. Available: https://aircconline.com/csit/papers/vol15/csit151407.pdf

[38] K. Balan, R. Learney, and T. Wood, "A Framework for Cryptographic Verifiability of End-to-End AI Pipelines," arXiv preprint arXiv:2503.22573, 2024.

[39] Z. Dehghani, "Data Mesh: Delivering Data-Driven Value at Scale," O'Reilly Media, 2022.

[40] A.Stage and D. Karastoyanova, "Trusted Provenance of Automated, Collaborative and Adaptive Data Processing Pipelines," arXiv preprint arXiv:2310.11442, 2023.

.

AUTHOR

Bala Subramanyan is a technologist and researcher with over 14 years of experience in secure systems architecture, privacy-preserving computation, and applied cryptography. He is the Co-Founder and CTO of Verifoxx, where he is the principal architect of a universal privacy infrastructure layer that leverages advanced PETs—including zero-knowledge proofs, verifiable computation, and trusted execution—to enable verifiable insights without exposing raw data. His work spans confidential computing, WebAssembly-based secure runtimes, and functional encryption. Bala's research and applied innovations have been featured in IEEE conferences and cryptographic forums such as IACR. Prior to co-founding Verifoxx, he led technical & R&D initiatives at JP Morgan, Lockheed Martin, Nationwide, and IHS, contributing to the design of scalable, proof-based systems for secure computation across finance, healthcare, and identity domains.