cWAMR: Reimagining A Capability-Based Webassembly Runtime Via CHERI-Based Compartmentalization

Bala Subramanyan

Verifoxx, London, UK

Abstract

WebAssembly (WASM) provides a language-neutral execution format widely deployed for cloud, edge, and embedded workloads. Its linear memory model and software-level sandboxing afford portability and a baseline of spatial isolation, yet they rely on dynamic bounds checks and page-based protections that incur overhead and remain susceptible to violations under speculative and out-of-order execution. Prior efforts to secure WASM for untrusted workloads frequently embed it within Trusted Execution Environments (TEEs) such as Intel SGX, introducing attestation, enclave management complexity, and exposure to shared-cache side channels, while still lacking hardware-enforced pointer provenance and bounds.

*This work presents **cWAMR**, the first WebAssembly runtime ported to leverage CHERI's hardware-enforced capability model, integrating fine-grained bounds, permissions, and pointer provenance directly into the execution of WASM modules. We describe the adaptations made to the WAMR runtime, including a CHERI-sealed memory allocator, capability-restricted system interface (cWASI), and secure externref handling, enabling WASM workloads to execute within CHERI compartments without reliance on enclave-wide isolation boundaries. Validation on the Arm Morello CHERI platform demonstrates correct execution of AoT-compiled and interpreted WASM modules under capability enforcement, preserving memory safety, compartmentalization, and integrity guarantees throughout runtime operation. Developed under the UK Digital Security by Design (DSbD) CHERI Morello program, this work establishes a practical path for integrating hardware capability systems with portable runtime environments. It lays the groundwork for future toolchain support, performance characterization, and broader deployment of capability-based security for untrusted code execution.*

Keywords

WebAssembly Runtime, Capability-Based Security, CHERI, Memory Safety, Compartmentalization

1. Introduction

1.1. Motivation

WebAssembly (WASM) has evolved into a widely adopted, language-neutral execution format for deploying software across heterogeneous platforms—including serverless Function-as-a-Service (FaaS), edge computing nodes, IoT devices, and cloud-native microservices. By compiling high-level languages such as C, C++, and Rust into a compact bytecode based stack machine with a linear memory abstraction, WASM offers portability, fast instantiation times, and software-level sandboxing. Standardized interfaces like the WebAssembly System Interface (WASI) [14] extend WASM's reach by enabling platform-neutral access to files, clocks, random generators, and emerging features like networking and HTTP support—making it practical well beyond browser contexts.

However, the security guarantees of WASM remain fundamentally software-bound. Its linear memory model is protected by dynamic bounds checks and virtual memory mechanisms (e.g., guard pages with ***mprotect***), [33] which impose significant performance overhead—especially evident in compute-intensive workloads—and still leave room for out-of-bounds memory accesses under speculative or transient execution. Prior evaluations across runtimes and architectures have shown that WASM's bounds-checking can degrade performance by up to 20%–650% [33].

To mitigate these risks, practitioners have explored running WASM inside hardware-backed Trusted Execution Environments (TEEs) such as Intel SGX [9] or AMD SEV. While TEEs encrypt memory regions and authenticate workloads to remote parties, they introduce substantial enclave transition costs, EPC paging overheads, and persistent vulnerability to microarchitectural side channels. Moreover, TEEs enforce coarse-grained trust boundaries, lacking the fine-grained per-object memory integrity needed to compartmentalize individual WASM modules or data structures.

1.2. Objectives and Contributions

This landscape motivates a shift toward enforcing memory safety and compartmentalization directly at the architectural level, without relying solely on cryptographic boundaries or runtime-managed checks.

CHERI (Capability Hardware Enhanced RISC Instructions) [1][2] embodies such a shift. By extending processor architectures to natively support unforgeable, metadata-rich pointers—capabilities that embed bounds, permissions, and provenance—CHERI enforces spatial and temporal safety guarantees in hardware, preventing unauthorized memory access even under speculation [34].

In this paper, we introduce cWAMR [24], the first integration of a mature WebAssembly runtime with the CHERI architecture. Specifically:

- We adapt the WebAssembly Micro Runtime (WAMR) [13] to produce and operate on CHERI capabilities, including a sealed capability memory allocator, a capability-enforced WASI interface (cWASI), and secure externref handling.
- We demonstrate that WASM modules can run inside hardware-enforced CHERI compartments, eliminating the need for enclave-wide encrypted memory and heavy context transitions, yet still maintaining strict bounds, provenance, and isolation guarantees.
- We validate our runtime on the Arm Morello platform [23], showing successful execution of AoT-compiled and interpreted WASM workloads under CHERI's security model.

These contributions represent a first step toward bringing architectural capability security to language-neutral, high-level runtime ecosystems—a direction aligned with recent priorities identified in national software security initiatives.

1.3. Paper Organization

The remainder of this paper is structured as follows:

- **Section 2** reviews key background on WASM runtimes, TEEs, and the CHERI architecture, and surveys related double-sandbox models.
- **Section 3** presents the cWAMR architecture, describing how CHERI capabilities were integrated into WAMR to achieve hardware-enforced memory safety and compartmentalization.

- **Section 4** details implementation, including CHERI-specific adaptations to memory allocation, execution contexts, and AoT toolchain workflows.
- **Section 5** validates cWAMR on Arm Morello, confirming secure execution under CHERI, and outlines ongoing efforts in toolchain development and benchmarking.
- **Section 6** concludes with insights on advancing capability-based secure execution.

2. BACKGROUND AND RELATED WORK

2.1. WebAssembly and Language Agnostic Runtimes

WebAssembly (WASM) [6][19] is a low-level binary instruction format designed as a portable compilation target for high-level languages such as C, C++, and Rust. Unlike conventional virtual machines like the JVM or CLR—which maintain rich object models and garbage collection—WASM implements a minimalist stack machine coupled with a linear memory abstraction: a module’s entire heap is represented as a single, contiguous, byte-addressable array. Programs use untyped integer offsets to index into this space.

This architecture minimizes runtime complexity by offloading memory safety to explicit checks rather than implicit object models or hardware-assisted segmentation. It also decouples execution from any single instruction set architecture, allowing WASM binaries to run unchanged across diverse platforms.

Security in WASM is primarily enforced via software sandboxing. Each module is isolated in its own linear memory, with no capability to perform arbitrary system calls or access external memory directly. Instead, interactions with the host system—whether for file I/O, networking, or cryptographic operations—are explicitly mediated by interfaces such as the WebAssembly System Interface (WASI) [14]. WASI provides a standardized, platform-neutral set of APIs, effectively acting as a syscall layer under tight control of the runtime.

However, WASM’s reliance on dynamic bounds checks and virtual memory protections leaves it vulnerable in adversarial contexts. Studies such as *“Leaps and Bounds”* [33] have demonstrated that bounds-checking overhead can become a dominant factor in runtime performance, and that speculative or out-of-order execution may transiently bypass these software checks. This exposes memory contents that ought to be logically inaccessible, undermining both confidentiality and integrity—especially in multi-tenant or untrusted execution environments.

2.2. Trusted Execution Environments (Tee)

Trusted Execution Environments (TEEs) provide hardware-isolated enclaves that shield code and data from the operating system and hypervisors. Prominent examples include Intel Software Guard Extensions (SGX) [11], AMD Secure Encrypted Virtualization (SEV) [17], and AWS

Nitro Enclaves [16]. These environments establish a cryptographic trust boundary to protect sensitive computations from a potentially compromised host.

Intel SGX [11] partitions physical memory into protected Enclave Page Cache (EPC) regions, ensuring that data inside the enclave remains encrypted outside the CPU package. SGX supports remote attestation, enabling cloud clients to verify enclave integrity before provisioning secrets.

However, SGX's design incurs several bottlenecks [25][32]:

- **OCALL/ECALL overhead:** System calls and external library invocations require explicit transitions across the enclave boundary, involving argument marshalling, encryption, and validation.
- **EPC paging penalties** [36]**:** SGX reserves a dedicated Enclave Page Cache (EPC), typically 128 MB to 256 MB per socket, determined by BIOS configuration at boot. While SGX2 extends the instruction set to dynamically allocate or free EPC pages within an enclave, it does not expand the total EPC size. Oversubscribed EPC workloads trigger encrypted paging to untrusted DRAM, protected by integrity trees, which imposes substantial latency penalties.
- **Side-channel vulnerabilities:** Despite memory encryption, shared microarchitectural resources (caches, branch predictors) can be exploited via transient execution attacks such as Spectre [7], Meltdown [8], and LVI [26], allowing leakage of enclave secrets.

AMD SEV [17] encrypts the entire guest VM's memory, isolating it from a malicious hypervisor. While this provides strong confidentiality, SEV operates at the VM granularity, lacking fine-grained compartmentalization within applications. Similarly, AWS Nitro Enclaves [16] spin off lightweight VMs with dedicated CPUs and memory, suitable for secure workload separation but too coarse for per-module or intra-process isolation.

In all these designs, TEEs elevate the trust boundary and provide assurances against external threats, but they do not alter the semantics of pointers or enforce bounds at the hardware level. Once inside an enclave, applications still rely on conventional memory safety mechanisms, which can be subverted by programming bugs or speculative execution side channels.

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.

Formally verified proofs of CHERI's architecture confirm that under typical attacker models—including speculative execution [34] where transient violations must resolve to legitimate architectural state—capabilities uphold bounds, provenance, monotonicity, and encapsulation properties. This moves the trust model from external cryptographic wrappers (like enclave encryption) to intrinsic, per-pointer hardware checks, eliminating classes of spatial and temporal vulnerabilities.

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 sandbox boundaries. These studies confirm that while WASM enforces spatial safety via software-managed mechanisms, it remains sensitive to speculative leaks and incurs notable performance tradeoffs under memory protection.

The WebAssembly Micro Runtime (WAMR) [13], on which our work builds, offers interpretation, AoT, and JIT modes optimized for constrained devices and standalone deployments. However, WAMR still relies on conventional linear memory bounds checks and does not enforce architectural provenance or bounds directly at the instruction set level.

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 do not redefine the semantics of memory references—pointers inside enclaves 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.

Recent CHERI-focused studies have primarily addressed OS kernels, native applications, and mitigation of speculative attacks via architectural speculation contracts [34]. 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. By doing so, cWAMR sidesteps the high transition costs and coarse compartment models of TEEs, while directly leveraging CHERI's architectural guarantees to confine untrusted WASM workloads.

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.

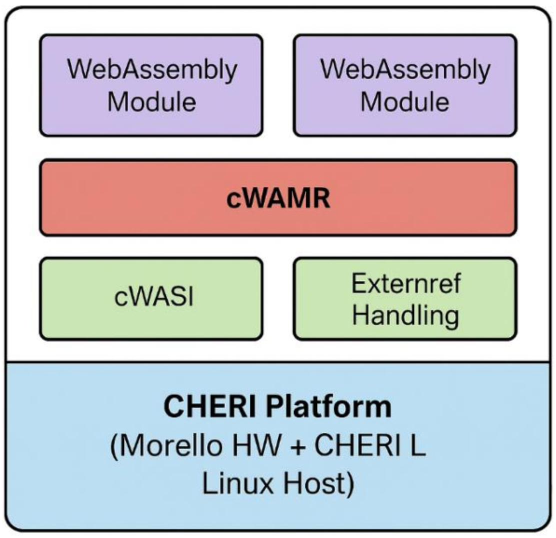

Figure 1. Layered Architecture of cWAMR [24] on the CHERI Platform

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 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]).

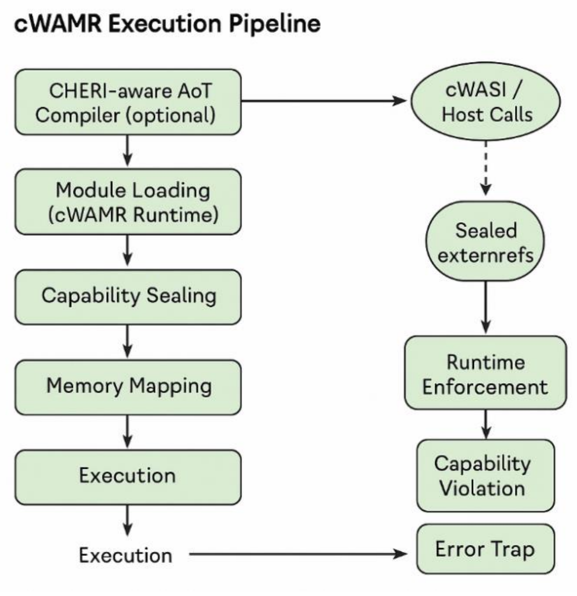

Figure 2. cWAMR Execution Pipeline Overview

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.

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.

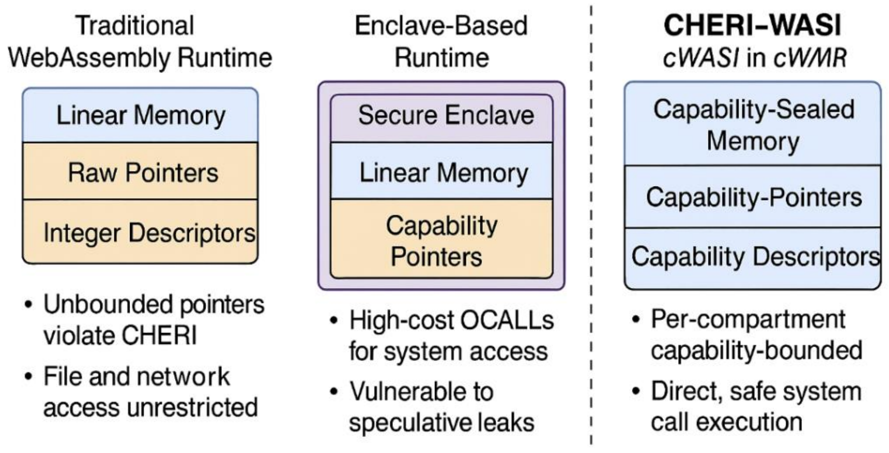
Traditional WebAssembly Runtime
Linear Memory
Raw Pointers
Integer Descriptors
• Unbounded pointers violate CHERI
• File and network access unrestricted
Enclave-Based Runtime
Secure Enclave
Linear Memory
Capability Pointers
• High-cost OCALLs for system access
• Vulnerable to speculative leaks
CHERI-WASI
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.

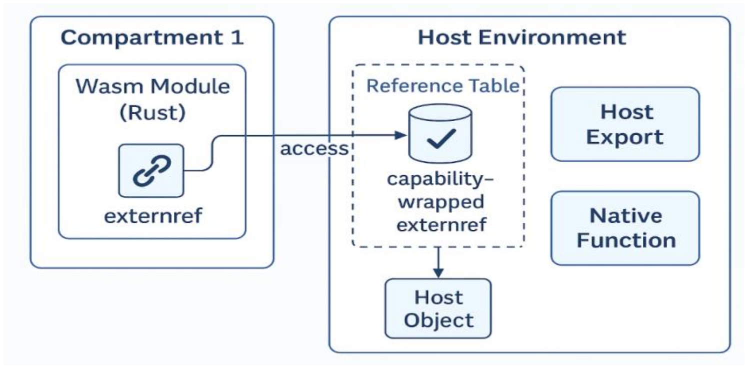

Figure 4. Secure wasm interop using capability-wrapped externrefs

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.

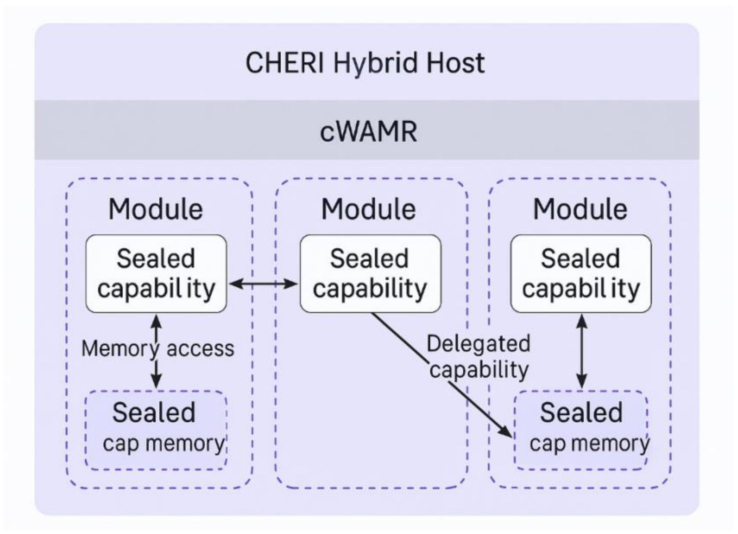

Figure 5. Per-Module Isolation and Capability Delegation in cWAMR

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.

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

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.

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 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

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 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.

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.

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 high-assurance, capability-native WebAssembly execution, bridging open standards (WASI, WAMR) with hardware-enforced security guarantees via CHERI.

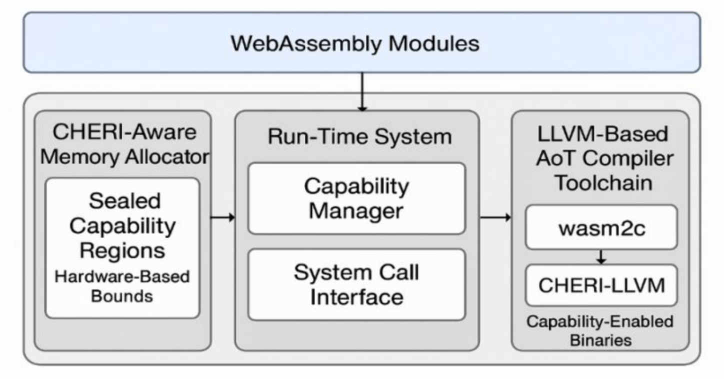
WebAssembly Modules
CHERI-Aware Memory Allocator
Sealed Capability Regions
Hardware-Based Bounds
Run-Time System
Capability Manager
System Call Interface
LLVM-Based AoT Compiler Toolchain
wasm2c
CHERI-LLVM
Capability-Enabled Binaries

Figure 6. Implementation

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 Capability Enforcement

Through interactive testing and coverage inspection, we've confirmed at these initial stages:

- WASM modules executed correctly under CHERI enforcement, both in hybrid and purecap configurations.
- Capability traps were triggered as expected when attempting to access out-of-bounds memory or dereference forged pointers.
- Memory access integrity was preserved across all benchmark workloads using sealed and bounded CHERI capabilities.
- No raw pointer dereferencing occurred during system call handling due to integration with the capability-aware CHERI-WASI layer.

These results validate that:

- The porting of WAMR to CHERI is successful and stable.
- WASM modules are able to operate securely under CHERI's architectural constraints.

5.2. Next Steps

With the current implementation, cWAMR [24] 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.
- **Deeper compiler pipeline integration:** refining AoT support with CHERI-LLVM to minimize sealing/unsealing overhead and ensure fine-grained capability propagation.
- **Enhanced toolchain automation:** improving reproducibility and developer ergonomics with robust presets, debug tooling, and portable CI pipelines for both Morello hardware and CHERI-QEMU [23] emulation.
- **Security explorations beyond memory safety:** such as integrating with architectural speculation contracts (e.g., CHERI CSC) [34] to evaluate transient execution containment.

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**, the first WebAssembly runtime ported to natively leverage CHERI's hardware-enforced capability system. By embedding fine-grained bounds, permission checks, and provenance directly into WASM execution semantics, cWAMR transitions away from reliance on software-only sandboxing and enclave-centric trust models—enabling least-privilege, hardware-backed isolation for untrusted code.

Our port demonstrates that CHERI's compartmentalization primitives can secure both WASM linear memory and its external interfaces, without enclave exit overhead, or coarse VM boundaries. Functional validation on CHERI Morello confirms correct capability propagation, trap behavior on violations, and stable execution of AoT and interpreted WASM modules.

As an open-source contribution under the UK Digital Security by Design (DSbD) initiative, cWAMR establishes a practical foundation for deploying secure-by-construction WebAssembly runtimes. Future work will expand toward systematic performance characterization, enhanced compiler-level optimizations, and security explorations — driving cWAMR toward a robust ecosystem complete with streamlined toolchain support, developer-friendly workflows, and rigorous security assurances — enabling practical, high-assurance WebAssembly deployments for privacy-preserving and multi-tenant computing.

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] Waterman, A., & Asanović, K. (2019). RISC-V Instruction Set Manual Volume I: User-Level ISA.

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

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

[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 (PLDI),* 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 *externref*, Rust documentation. [Online]. Available: https://docs.rs/externref/latest/externref/

[29] 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

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

[31] 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.

[32] 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

[33] 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."

[34] 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.

[35] 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

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 collaborative analytics without exposing raw data. His work spans confidential computing, TEEs, 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.