TUDelft

Delft University of Technology

Cage

Hardware-Accelerated Safe WebAssembly

Fink, Martin; Stavrakakis, Dimitrios; Sprokholt, Dennis; Chakraborty, Soham; Ekberg, Jan Erik; Bhatotia, Pramod

Citation (APA)
Fink, M., Stavrakakis, D., Sprokholt, D., Chakraborty, S., Ekberg, J. E., & Bhatotia, P. (2025). Cage: Hardware-Accelerated Safe WebAssembly. In J. Doerfert, T. Grosser, H. Leather, & P. Sadayappan (Eds.), *CGO 2025 - Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (pp. 538-552). (CGO 2025 - Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization). Association for Computing Machinery (ACM). https://doi.org/10.1145/3696443.3708920

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

CAGE: Hardware-Accelerated Safe WebAssembly

Martin Fink
martin.fink@cit.tum.de
Technical University of Munich
Munich, Germany

Dimitrios Stavrakakis
dimitrios.stavrakakis@tum.de
Technical University of Munich
Munich, Germany

Dennis Sprokholt
d.g.sprokholt@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Soham Chakraborty
S.S.Chakraborty@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Jan-Erik Ekberg
jan.erik.ekberg@huawei.com
Huawei Technologies
Helsinki, Finland

Pramod Bhatotia
pramod.bhatotia@tum.de
Technical University of Munich
Munich, Germany

Abstract

WebAssembly (WASM) is an immensely versatile and increasingly popular compilation target. It executes applications written in several languages (e.g., C/C++) with near-native performance in various domains (e.g., mobile, edge, cloud). Despite WASM's sandboxing feature, which isolates applications from other instances and the host platform, WASM does not inherently provide any memory safety guarantees for applications written in low-level, unsafe languages.

To this end, we propose CAGE, a hardware-accelerated toolchain for WASM that supports *unmodified* applications compiled to WASM and utilizes diverse Arm hardware features aiming to enrich the memory safety properties of WASM. Precisely, CAGE leverages Arm's Memory Tagging Extension (MTE) to *(i)* provide spatial and temporal memory safety for heap and stack allocations and *(ii)* improve the performance of WASM's sandboxing mechanism. CAGE further employs Arm's Pointer Authentication (PAC) to prevent leaked pointers from being reused by other WASM instances, thus enhancing WASM's security properties.

We implement our system based on 64-bit WASM. We provide a WASM compiler and runtime with support for Arm's MTE and PAC. On top of that, CAGE's LLVM-based compiler toolchain transforms unmodified applications to provide spatial and temporal memory safety for stack and heap allocations and prevent function pointer reuse. Our evaluation on real hardware shows that CAGE incurs minimal runtime (< 5.8 %) and memory (< 5.3 %) overheads and can improve the performance of WASM's sandboxing mechanism, achieving a speedup of over 5.1 %, while offering efficient memory safety guarantees.

CCS Concepts: • **Security and privacy → Systems security**; • **Software and its engineering → Compilers**.

Keywords: WebAssembly, Memory Safety

1 Introduction

WebAssembly (WASM) [23] has been gaining prominence as a versatile compilation target [38]. WASM allows for deploying and executing native applications written in a variety of languages, such as C/C++ and Rust, in a wide spectrum of environments (e.g., Web, Edge Cloud, IoT devices) [5], while achieving near-native performance. In principle, WASM enables the compilation of high-level languages (e.g., C/C++) to its bytecode format, which can then be seamlessly compiled to native machine code based on the targeted underlying architecture, which justifies its increasing adoption.

A core design principle of WASM is the *sandboxing* of untrusted code [4], providing a safety property of Software Fault Isolation (SFI) [54]. Each WASM application is confined in its isolated address space and has no access rights beyond that. Thus, WASM protects the host and other guest WASM instances from potentially malicious or buggy code.

However, applications compiled to WASM are still vulnerable to memory safety issues, such as buffer overflows or dangling pointers, within an application's memory space, despite WASM's sandboxing [35]. Such issues allow attackers to manipulate the memory space of a WASM instance, corrupting or leaking sensitive data or manipulating the control flow. This limitation becomes particularly evident when compiling applications written in memory-unsafe languages, e.g., C/C++. Importantly, numerous CVEs remain exploitable when compiled to WASM (§3).

Unfortunately, existing memory safety solutions, designed for applications compiled to native machine code, cannot be directly applied for WASM due to several factors:

– **Memory safe languages**, such as Rust or OCaml, tackle memory safety at the language level by encoding certain

M. Fink, D. Stavrakakis, D. Sprokholt, S. Chakraborty, J.-E. Ekberg, P. Bhatotia

properties in their type system or providing safe abstractions, e.g., in their standard libraries. However, it is not feasible to port all applications or libraries written in C/C++ to languages with such guarantees.

– **Trip-wire-based** memory safety approaches [12, 14, 24, 41, 47, 48] do not apply to WASM without altering the WASM memory layout properties, as these approaches rely on shadow memory or guard pages.

– **Pointer- or object-based** solutions [7, 15, 17, 18, 30, 31, 37, 39, 41] can be adapted to WASM, but they either modify the pointer layout and size, incur significant runtime overheads, or rely on custom hardware [55], thus, hindering production deployment in common WASM environments.

Hence, we aim to answer the question: *How can we design a practical system to provide memory safety properties for unmodified applications compiled to WASM without changing its linear memory model and introducing minimal overheads to allow for deployment in production environments?*

Our key insight is to leverage modern commodity hardware extensions by designing abstractions for WASM that can be directly used by the compilers. Precisely, Arm recently introduced new ISA extensions, namely PAC [46] and MTE [11], that can serve as a base to design practical, high-performance solutions against memory safety issues.

To this end, we propose Cage, a hardware-accelerated WASM toolchain that leverages Arm's MTE and PAC hardware extensions to provide both spatial and temporal memory safety issues for *unmodified* C/C++ programs. Cage further hardens applications against control flow highjacking in the form of function pointer reuse between WASM instances. Additionally, Cage improves the performance of the sandboxing mechanism for 64-bit WASM by offloading the bounds checks to MTE hardware. We design Cage so it can fall back to software-based implementations on devices lacking the relevant hardware.

To the best of our knowledge, Cage is the *first* practical, comprehensive, and efficient solution for unmodified C/C++ programs that can be deployed on commodity hardware and provides strong memory safety guarantees for WASM instances running memory-unsafe code on Arm platforms.

Altogether, Cage makes the following contributions:

– **WebAssembly extension:** A minimal and generic extension to the WebAssembly specification to provide memory safety guarantees and is deployable on every platform, regardless of the availability of specialized hardware.

– **Compiler toolchain:** A compiler toolchain that transparently hardens unmodified programs to enforce spatial and temporal memory safety for stack and heap allocations and prevent unintended function pointer reuse.

– **WASM runtime:** A hardware-accelerated WASM compiler and runtime, leveraging Arm's MTE and PAC with minimal overhead that can be deployed in production.
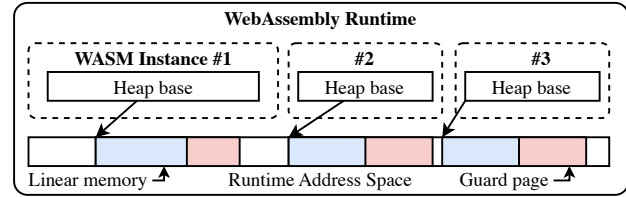

Figure 1. WASM's linear memory model.

– **Evaluation on real hardware:** Evaluation and security analysis of Cage's implementation on commercially available Arm hardware. Our evaluation is structured around performance and memory overheads and is accompanied by an extensive analysis of MTE and PAC performance as implemented on real hardware.

We implement our Cage prototype on top of 64-bit WASM, consisting of a compiler toolchain based on LLVM and a WASM compiler and runtime based on wasmtime incorporating support for Arm's MTE and PAC. We conduct an extensive analysis of Cage's security guarantees and evaluate Cage's performance using the PolyBench/C [45] benchmark suite. We further analyze Arm's MTE and PAC performance as implemented on production hardware through a set of microbenchmarks. Our evaluation on real Arm hardware shows that Cage provides its memory safety properties while incurring minimal runtime (< 5.6 %) and memory (< 5.3 %) overheads and is capable of significantly improving the performance of WASM's sandboxing feature, achieving a speedup of over 5.1 %.

2 Background

2.1 WebAssembly

WebAssembly [23] is a versatile, high-performance compilation target, initially designed as an alternative to JavaScript. Its goal is to execute applications written in any language with near-native performance regardless of the hardware and software stack of the hosting platform. Several high-level languages (e.g., C/C++, Rust) can be compiled to WASM's bytecode format. It is then lowered to the appropriate native machine code, depending on the underlying system architecture. This feature expands the usability of WASM to various other domains [5], such as Function-as-a-Service (FaaS) workloads or even as an alternative to Linux containers [1].

WASM employs a linear memory model (see Fig. 1). Thus, applications manage their memory without requiring unnatural idioms, and WASM runtimes can efficiently map the WASM instance memory directly to host memory. Importantly, WASM's design does not allow unstructured control flow. WASM uses indices into type- and bounds-checked tables instead of raw function pointers to make indirect function calls, while, for jumps, WASM provides a set of well-defined control flow constructs. Additionally, WASM
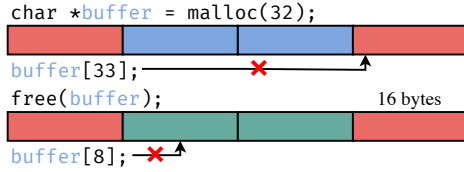
```
char *buffer = malloc(32);
```



```
buffer[33];
free(buffer);                    16 bytes
buffer[8];
```

Figure 2. Example of a heap allocation protected by MTE. After allocation, the pointer and allocation are tagged with ▮, while the surrounding memory is tagged with ▮. When accessing memory, the hardware catches out-of-bounds errors (▮ ≠ ▮). When freeing memory, the memory region is retagged with ▮. This prevents use-after-free errors (▮ ≠ ▮).

does not expose registers but operates on a verified, well-typed stack to ensure compatibility with diverse compilation targets that offer different sets of registers.

WASM also provides *sandboxing* for programs. The WASM runtime must ensure that each instance can only access memory within the bounds of its accessible linear memory. This is typically achieved using either *(i) explicit bounds checks* that are inserted before every memory access, validating that it lies within the WASM instance's memory range or *(ii) guard pages*, where 4 GiB of virtual memory is mapped and pages beyond the WASM instance's memory is marked as inaccessible, and any access in them results in a segmentation fault. This only works for 32 bit pointers, which don't allow accessing memory beyond 4 GiB. Typically, switching to 64-bit WASM entails switching to the more expensive approach *(i)* with explicit bounds checks.

2.2 Memory Safety

Applications written in low-level, memory-unsafe languages (e.g., C/C++) are prone to memory safety bugs that enable a whole class of attacks on a vulnerable or buggy program [50]. Several studies have shown that in large software projects, memory safety bugs make up between 70 % and 75 % of their security vulnerabilities [2, 52, 53]. These bugs are classified as *spatial* or *temporal*. Spatial memory safety errors occur when a memory access is performed beyond the allocated boundaries of a memory object (e.g., buffer overflows), while temporal memory safety bugs refer to accesses to memory regions before they are allocated or after their release (e.g., dangling pointers). These vulnerabilities can be exploited by attackers to overwrite data or redirect control flow, for instance, by manipulating the return address saved on the stack to create return-oriented programming (ROP) chains, chaining existing snippets of code together to create attacks.

To this end, several memory safety solutions have been proposed, which can be divided into three major categories: *(i) trip-wire-based approaches* [12, 14, 24, 41, 47, 48] that employ guard zones around memory objects and allocate designated memory regions, namely shadow memory, that determine whether a memory location is accessible or not, *(ii) object-based approaches* [7, 15, 17, 18] that track memory
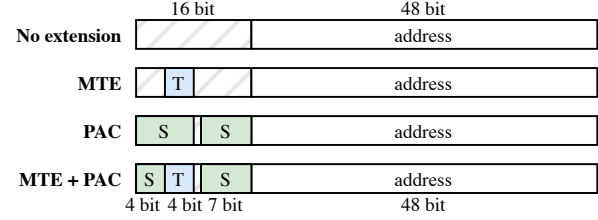

Figure 3. Pointer layout on aarch64 in Linux with and without MTE and PAC enabled.

safety metadata on a per-object level and ensure memory safety for pointers with respect to an object, and *(iii) pointer-based approaches* [13, 27, 30, 31, 39, 40, 42, 57] that keep track of object bounds by either storing them in the pointer itself (e.g., via fat pointers) or in external data structures. Additionally, specialized solutions [6, 32, 36] have also been proposed to mitigate control-flow attacks that occur via memory safety bugs and either analyze applications to enforce valid control flow paths or enforce memory safety for code pointers using hardware- or software-based techniques.

Memory tagging approaches [11, 49] combine aspects from object- and pointer-based approaches. They typically associate metadata stored in the unused bits of a pointer with memory objects by assigning tags to allocated memory regions and performing checks at runtime.

2.3 Memory Safety Hardware Extensions

As an alternative to flexible yet slow software-based memory safety solutions, CPU designers develop hardware extensions [11, 13, 42, 46] to serve as a foundation for efficient memory safety. They provide security primitives to fortify applications with memory safety properties while having minimal memory and performance overheads, making them suitable for production deployment.

Fig. 3 presents an example layout of a pointer on aarch64, the 64-bit variant of the ArmV8 ISA [10]. Only 48 out of the available 64 bits are used to address memory. The remaining bits are set to 0 or 1 to differentiate between kernel and userspace. Hardware extensions such as Top Byte Ignore (TBI), MTE (§2.3), or PAC (§2.3) leverage those unused bits for storing metadata.

Memory Tagging Extension (MTE). ARMs MTE provides a building block for spatial and temporal memory safety [11]. MTE implements a lock-and-key mechanism where memory regions can be tagged with one of 16 distinct tags, and memory accesses are only allowed using pointers with the corresponding tags. The locking mechanism stores a 4-bit tag in bits 56–59 of an address. Accordingly, a tag is assigned to memory with a granularity of 16 bytes.

Fig. 2 presents how MTE can provide *spatial* as well as *temporal* memory safety. Precisely, MTE can ensure spatial memory safety by assigning different tags to adjacent memory regions, while it also can offer temporal memory safety by retagging freed memory regions.

Table 1. MTE and PAC instruction throughput (instructions per cycle, higher is better) and latencies (cycles, lower is better). We only show PAC instructions using the Data A-key (da).

Inst	Cortex-X3		Cortex-A715		Cortex-A510	
	Tp	Lat	Tp	Lat	Tp	Lat
MTE						
irg	1.34	1.99	1.00	2.00	0.50	3.00
addg	2.01	1.99	3.81	1.00	2.22	2.00
subg	2.01	1.99	3.81	1.00	2.22	2.00
subp	3.49	0.99	3.81	1.00	2.50	2.00
subps	2.88	0.99	3.80	1.00	2.50	2.00
stg	1.00	–	1.81	–	1.00	–
st2g	1.00	–	1.84	–	0.46	–
stzg	1.00	–	1.84	–	0.98	–
st2zg	0.34	–	1.79	–	0.45	–
stgp	1.00	–	1.69	–	0.98	–
ldg	2.92	–	1.91	–	0.93	–
PAC						
pacdza	1.01	4.97	1.51	5.00	0.20	4.99
pacda	1.01	4.97	1.42	5.00	0.20	5.00
autdza	1.01	4.97	1.51	5.00	0.20	7.99
autda	1.01	4.97	1.43	5.00	0.20	7.99
xpacd	1.01	1.99	1.56	2.00	0.20	4.99

MTE currently can be set in one the following modes: *(i) disabled*, where no tag checks are performed, *(ii) synchronous*, where a tag mismatch immediately triggers a fault disallowing the read/write at the affected memory location, *(iii) asynchronous*, where a tag mismatch does not cause a fault directly but sets a CPU flag that is checked at the next context switch, thus allowing for a potential read/write at the affected memory location by the triggering command, and *(iv) asymmetric*, where reads are checked asynchronously and writes are checked synchronously.

Pointer Authentication (PAC). PAC [46] introduces hardware primitives to prevent attackers from forging pointers. PAC places a 7 to 16 bit signature in the upper bits of pointers, with the exact layout being dependent on the operating system, the underlying hardware, and other factors (e.g. if MTE is enabled). Signatures are created using the pointer value, a secret key placed in an inaccessible register, and a user-defined value (modifier). Signed pointers cannot be used directly to access memory; they must be authenticated. Authenticating a pointer consists of validating the signature and stripping out the signature if the validation is successful, thus producing a valid pointer. In case of a failed authentication, PAC can either produce a pointer that will trap on memory access or trap immediately. This behavior depends on whether FEAT_FPAC is implemented [10].

MTE and PAC can be combined at the cost of bits available for the PAC signature. The exact layout of the PAC signature varies depending on the system. On Linux, bits 56−59 are used for MTE while bits 63−60 and 54−49 are used for PAC, as shown in Fig. 3. The remaining bit 55 differentiates between kernel- and user-space addresses.
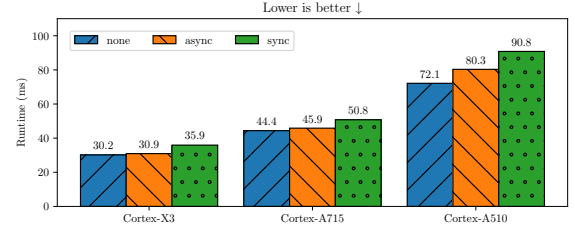

Figure 4. Performance overhead of MTE sync and async mode for writing 128 MiB of memory. See §7.1 for details on the experimental setup.

Architectural performance analysis. We evaluate the performance characteristics of MTE as implemented on the Tensor G3 chip, such as throughput and latency of the individual instructions in Table 1. We run microbenchmarks executing 10^{10} instructions in an unrolled loop. To measure throughput, we execute the instructions without any data dependencies; to measure latency, we force a data dependency between subsequent instructions. For instructions storing and loading memory tags, we only measure throughput.

We further measure the raw overhead of enabling MTE. We perform a 128 MiB memset with MTE disabled, synchronous, and asynchronous mode. We perform each run with a clean cache. In Fig. 4, we observe that with synchronous MTE, memset is 19.1%, 14.4%, and 29.9% slower on the respective cores compared to the baseline with MTE disabled. Asynchronous MTE gets closer to the baseline with an overhead of 2.6%, 3.3%, and 11.3%, respectively.

3 Motivation

WebAssembly is inherently protected by design against a wide range of attack vectors, such as jumping to arbitrary addresses or injecting shellcode. Despite its protection mechanisms, it is shown that WebAssembly is still susceptible to attacks originating from memory safety issues, such as buffer overflows or dangling pointer accesses [35].

Importantly, WASM compilers place data in the linear memory of the WASM instance, where both read and write permissions are granted to the executed code. The lack of read-only memory regions and the ability to map arbitrary pages in WASM prevents measures, such as Address Space Layout Randomization (ASLR) or guard pages, from being applied. Thus, in case of a successful exploit of a memory safety bug, an attacker can overwrite application data. Table 2 presents a set of exploits that were discovered outside the scope of WASM but serve as examples that showcase potential memory safety errors when the vulnerable applications run in a WASM instance.

Additionally, in WASM, a degree of control flow manipulation is possible. Precisely, function pointers can be overwritten with pointers to other functions that share the same signature. However, it can only occur with functions that are

---
[1] Although the ROP chain is mitigated, memory corruption is still possible.
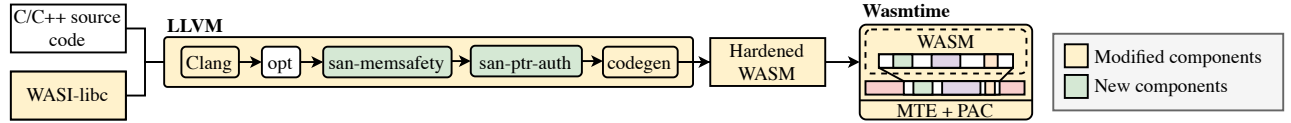
Figure 5. Overview of CAGE's components and instrumentation pipeline: Unmodified C/C++ code is compiled along with our modified WASI-libc. After optimizations, two sanitizer passes for memory safety and pointer authentication run, inserting new WASM instructions. Wasmtime processes this hardened WASM and emits MTE and PAC.

Table 2. An exemplary list of memory safety errors, their underlying cause, and the level of their mitigation.

CVE	Cause	Mitigated in WASM
CVE-2023-4863	Out-of-bounds	No
CVE-2014-0160	Out-of-bounds	No
CVE-2021-3999	Out-of-bounds	Partially[1]
CVE-2018-14550	Out-of-bounds	No
CVE-2021-22940	Use-after-free	No
CVE-2021-33574	Use-after-free	No
CVE-2020-1752	Use-after-free	No
CVE-2019-11932	Double-free	Partially[1]

present in the function table, i.e., functions that are targets of virtual function calls, such as virtual functions in C++. This results in a similar property as code-pointer separation (CPS) [32]. Listing 1 showcases such a vulnerability, as an attacker can overwrite a function pointer and redirect an indirect call to another function. When WASM is used in a WebOS-like scenario [29, 33, 56], i.e., running multiple instances in a single process, leaking function pointers in one program is even more vulnerable, especially if instances share a common library.

WebAssembly sandboxing. WASM engines use various techniques to protect their sandboxes against malicious code. While virtual memory and guard pages are preferred for performance reasons, some settings (e.g., 64-bit WASM) necessitate software-based bounds checks, which come at a higher performance cost. Based on our evaluation, switching from 32-bit to 64-bit WASM results in a roughly 6–8 % overhead on out-of-order CPUs, which can speculate through bounds checks, and 52 % overhead on in-order CPUs (§ 7.2). The measurements on out-of-order CPUs align with previous works [51]. The fallback to software-based bounds checks is especially painful when running on low-power in-order cores using 64-bit WASM or in environments without an OS, such as embedded devices. Additionally, software bounds checks or the guard pages technique may suffer from implementation bugs and must be protected against spectre-style attacks [28]. An example is CVE-2023-26489 [3], where an erroneous code lowering rule allowed malicious WASM instances to access memory outside the sandbox.

WASM is, despite limitations such as high overhead when running 64-bit WASM or the lack of a practical and low-overhead solution to prevent memory safety issues in programs written in C/C++, steadily growing in adoption [5].

```
struct VTable { void (*f)(); void (*g)(); };
void vulnerable(char *input) {
  struct VTable vtable = {.f = foo, .g = bar};
  char buf[16];
  strcpy(buf, input);
  vtable.f();
}
```

Listing 1. Vulnerable overflow allowing attackers to redirect control flow to call foo instead of bar.

Recently, manufacturers have started shipping hardware supporting both PAC and MTE, with the Google Pixel 8 being the first commercially available device to feature both. These extensions offer strong security guarantees with very low overhead, as shown in Table 1 and Fig. 4.

CAGE. To tackle the aforementioned memory safety issues, we design CAGE, an extension to WASM that efficiently prevents memory safety and function-pointer reuse exploits by leveraging Arm's MTE and PAC. On top of that, CAGE implements hardware-based sandboxing using MTE to circumvent the overhead of software-based checks and prevent vulnerabilities, e.g., CVE-2023-26489.

4 Design

CAGE consists of a WASM extension to provide memory safety guarantees within a WASM instance and improve the performance of the sandboxing mechanism. Its core design principles are to be *(i)* minimally invasive and *(ii)* applicable on diverse platforms using various approaches, including hardware extensions, such as MTE and PAC, software-based techniques, or hybrid solutions similar to HWASan [49].

Figure 5 presents an overview of CAGE. Precisely, unmodified C/C++ source code is compiled using the LLVM toolchain [34], where CAGE includes a sanitizer that instruments stack allocations and function pointers, along with a modified libc based on WASI [22] that protects heap allocations. LLVM's backend generates the hardened WASM binaries that can be executed in wasmtime [9], which implements our extension using MTE and PAC.

4.1 System Model

Threat model. Figure 6 highlights two aspects of memory safety, present in WASM, namely *internal* and *external* memory safety. We depict trusted components (⊘) in green and untrusted components (⊗) in red. We further annotate the hardened component by CAGE in each of these models (⊖).

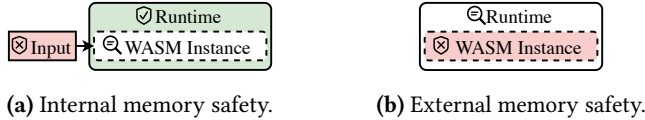**(a)** Internal memory safety. **(b)** External memory safety.

Figure 6. Internal and external memory safety in WASM.

The *internal memory safety* model (Fig. 6a) mirrors the threat environment of a standard non-WASM program. In this case, the application in the sandbox and its runtime, including the compiler, are considered trusted and assumed to be bug-free. Untrusted input (e.g., network data) originates outside the sandbox and can be controlled by an attacker. This implies that common memory safety bugs, such as buffer overflows or use-after-free, can be exploited to tamper with the WASM memory. WebAssembly's design inherently mitigates some threats common in non-WASM environments (§ 2.1), so we do not consider ROP-style attacks or those relying on unstructured control flow.

The *external memory safety* model (Fig. 6b) refers to the security of the sandbox. Threats originate from running untrusted programs, which may be adversarial or contain bugs. Typical attacks include sandbox escapes, where an attacker attempts to break out of the sandbox's restrictions and access host resources, or side-channel attacks, where attackers exploit timing differences or resource usage patterns to infer sensitive information. In this setting, we assume that the operating system and the underlying architecture do not have exploitable bugs. Importantly, we do not make assumptions about potential spectre-like [28] attacks. We do not protect against protected against side-channel attacks.

Programming model. CAGE currently supports unmodified applications written in C/C++ that target 64-bit WASM. Note that our toolchain is not bound to any language and can handle other languages compiled to LLVM in the future. CAGE provides spatial and temporal heap safety to applications that use its adapted WASI-libc that comes with a hardened allocator. For applications using their own allocator, we expose CAGE's memory safety primitives to C, enabling programmers to implement the same security guarantees. Lastly, CAGE reserves the unused upper 16 bits of 64-bit pointers to place memory safety metadata.

Deployment model. CAGE's prototype is highly optimized to use Arm's hardware extensions to minimize the runtime overheads. Therefore, its primary deployment target is Arm CPUs with Arm PAC and MTE extensions. However, CAGE can also be deployed on any platform, regardless of the underlying hardware, where the respective memory safety protection mechanisms have an equivalent software fallback, resulting in higher performance overheads.

4.2 WebAssembly Extension

CAGE essentially is an extension to WebAssembly that introduces primitives that can be used by a modified standard library or sanitizers to provide memory safety guarantees

(new instructions) $e ::=$ **segment.new** o
           | **segment.set_tag** o
           | **segment.free** o
           | **i64.pointer_sign**
           | **i64.pointer_authenticate**

Figure 7. CAGE's new instructions: Segment instructions take a constant unsigned offset o, which allows compilers to fold in constant offsets when manipulating segments.
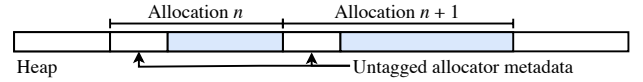
(a) Heap layout with untagged allocator metadata slots, ensuring no tag collisions for adjacent allocations.
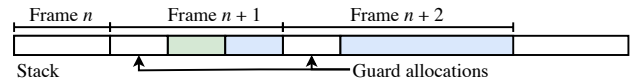
(b) Stack layout with guard slots ensuring tag collisions never occur for adjacent allocations.

Figure 8. Untagged slots serving as guard slots to prevent tag collisions for adjacent allocations.

for selected memory allocations. It builds on wasm64 [21], the 64-bit variant of WebAssembly. Wasm64 extends pointers within a WASM instance to 64 bits. Out of those, only 48 are used to index memory. This allows CAGE to utilize the remaining 16 bits to store its required memory safety metadata.

Memory safety. CAGE introduces the notion of *segments* and *tagged pointers* in the context of WASM. It further features five new instructions, shown in Fig. 7, that allow the creation of segments and the derivation of tagged pointers from raw pointers. The CAGE pointers carry provenance and can only access the segment they were created with. Conversely, segments can only be accessed by the tagged pointers created with them.

At an instance startup, the linear memory consists of a single segment that can be accessed via untagged indices, allowing unmodified code to run under our new semantics without modifications. This design choice allows the gradual integration of safety primitives into specific parts of WebAssembly applications where enhanced security is required. Thus, CAGE can achieve security properties similar to those of specialized hardware designed mainly for memory safety [57] by combining the concept of segments with Arm's hardware extensions (e.g., MTE).

Heap safety. In CAGE, the memory allocator must be aware of segments to provide heap safety. Adapting a memory allocator to utilize CAGE's memory safety extensions requires minimal modifications. When allocating memory, the allocator must align the requested size to 16 bytes, perform the allocation, and create a segment. The corresponding tagged pointer is then returned to the caller. CAGE randomly selects

Algorithm 1: Detect and harden safe and unsafe stack allocations.

Input : Allocations
Output: Hardened stack allocations
handle_stack_allocations(*allocations*)

begin
    allocsToInstrument ← ∅
    foreach *alloc* ∈ *allocations* **do**
        if *escapes(alloc)* **then**
            allocsToInstrument ← *allocsToInstrument* ∪ { *alloc* }
        else if *isUsedByUnsafeGEP(alloc)* **then**
            allocsToInstrument ← *allocsToInstrument* ∪ { *alloc* }
        end
    end
    foreach *alloc* ∈ *allocsToInstrument* **do**
        insertTaggingCode(*alloc*)
        insertUntaggingCode(*alloc*)
    end
    if *allocsToInstrument* ≠ ∅ ∧ *allocations*[0] ∉ *allocsToInstrument* **then**
        insertGuardAlloc()
    end
end

---

a tag for each allocation. When freeing memory, CAGE's allocator uses the `segment.free` instruction that ensures the detection of potential use-after-free and double-free errors.

Further, CAGE has to ensure that there are no tag collisions for adjacent allocations to provide protection against off-by-one buffer overflows/underflows. This is achieved by design as CAGE places metadata at the beginning of every heap allocation, and the corresponding memory region is preserved untagged, as shown in Fig. 8a. Thus, adjacent allocations are always separated by an untagged memory segment.

Stack safety. To provide memory safety for the stack, CAGE creates segments for the stack slots when entering a function. CAGE generates a random tag per function for the first stack allocation. Subsequent stack allocations use this tag and increment it by one. As the available tag bits are limited, the tag wraps around on overflow. Before returning from a function, all stack slots are untagged and reassigned to the original stack frame. This allows other functions to use the memory and prevents stack slots from being accessed after returning from a function. Note that each stack allocation needs to be aligned to 16 bytes and gets processed when entering/exiting from a function.

Similarly to the heap, CAGE must guarantee protection for off-by-one overflows/underflows on the stack. Therefore, CAGE must ensure that two adjacent stack allocations between functions do not share the same tag. To achieve this, CAGE inserts a single untagged stack guard slot at the beginning of the frame if no such untagged stack slot exists, as shown in Fig. 8b. Without the guard allocation, adjacent allocations in stack frames $n + 1$ and $n + 2$ would share the same tag (blue) and, thus, CAGE would not be able to detect overflows.

To further reduce its performance and memory overheads, CAGE omits the instrumentation of stack allocations that *(i)* do not escape the function or *(ii)* are only accessed using statically verifiable indices. In Algorithm 1, we present a simplified version of CAGE's algorithm for the identification of the (un)safe stack allocations.
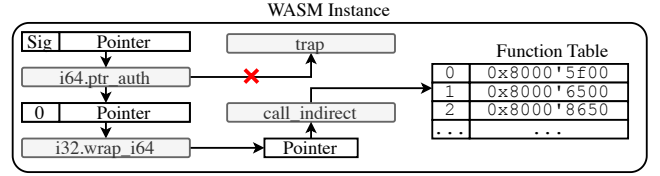

Figure 9. Our modified instruction sequence for indirect function calls.

$$\frac{C_{\text{memory}} = n}{C \vdash \textbf{segment.new } o : \text{i64 i64} \rightarrow \text{i64}}$$

$$\frac{C_{\text{memory}} = n}{C \vdash \textbf{segment.set\_tag } o : \text{i64 i64 i64} \rightarrow \epsilon}$$

$$\frac{C_{\text{memory}} = n}{C \vdash \textbf{segment.free } o : \text{i64 i64} \rightarrow \epsilon}$$

$$\frac{}{C \vdash \textbf{i64.pointer\_sign} : \text{i64} \rightarrow \text{i64}}$$

$$\frac{}{C \vdash \textbf{i64.pointer\_auth} : \text{i64} \rightarrow \text{i64}}$$

Figure 10. Typing rules of CAGE's new instructions. For the definition of context C, see the WASM paper [23].

Pointer authentication. CAGE provides pointer authentication primitives that prevent function pointer reuse between WASM instances. On the instantiation of a WASM module, a secret key is generated. The key is not accessible by the user code. CAGE's authentication operations leverage this key to sign and authenticate pointers using a cryptographic hash function. The signature is placed in the unused 16 bits of a WASM pointer, alongside the pointer tag, if applicable. Pointers containing a signature cannot directly access memory. On authentication, the signature is checked and stripped, if it is valid. Otherwise, the WASM module traps. While function pointer reuse within a WASM instance is still possible, CAGE prevents the reuse across different instances, as each instance generates its own key.

While memory64 [21] extends function pointers to 64 bits, the indices for the WASM function table remain 32 bit wide. CAGE uses the instruction sequence in Fig. 9 to authenticate function pointers and perform indirect function calls. 64-bit pointers are first authenticated, which traps in case of a invalid signature. If successful, the signature is stripped and the pointer is truncated to 32 bits. Similarly, when creating function pointers, indices into the function table are first zero-extended to 64 bits and then signed.

5 Semantics

5.1 Typing Rules

CAGE extends the typing rules of the original WASM paper [23], as shown in Figure 10. We adopt the notation used in previous work [43]. Specifically, the rules are of the form of $C \vdash e : tf$. An instruction e is valid under the context C, with C_{memory} being used to access a context component, such as the memory. The rule $C_{\text{memory}} = n$ ensures that the

$$\text{(store) } s ::= \{\ldots, \text{inst } inst^*\}$$
$$inst ::= \{\ldots, \text{tag } taginst^*, \text{key } k_s\}$$
$$taginst ::= b^*$$

$$s; (\textbf{i64.const } k) \ (t.\textbf{load } a \ o) \ \hookrightarrow_i \ (t.\textbf{const } b^*) \qquad \text{if } s_{\text{tag}}(i, k + o, |t|) = \text{tag}(k) \wedge s_{\text{mem}}(i, k + o, |t|) = b^* \quad (1)$$
$$s; (\textbf{i64.const } k) \ (t.\textbf{load } a \ o) \ \hookrightarrow_i \ \textbf{trap} \qquad \text{otherwise} \quad (2)$$
$$s; (\textbf{i64.const } k) \ (t.\textbf{const } c) \ (t.\textbf{store } a \ o) \ \hookrightarrow_i \ s'; \epsilon \qquad \text{if } s_{\text{tag}}(i, k + o, |t|) = \text{tag}(k) \quad (3)$$
$$\wedge \ s' = s \text{ with } \text{mem}(i, k + o, |t|) = \text{bits}_t^{|t|}(c)$$
$$s; (\textbf{i64.const } k) \ (t.\textbf{const } c) \ (t.\textbf{store } a \ o) \ \hookrightarrow_i \ \textbf{trap} \qquad \text{otherwise} \quad (4)$$
$$s; (\textbf{i64.const } k) \ (\textbf{i64.const } l) \ (\textbf{segment.new } o) \ \hookrightarrow_i \ s'; (\textbf{i64.const } t) \qquad \text{if } t = \text{new\_tag}(k + o) \wedge s' = s \text{ with } \text{tag}(i, k + o, l) = t \quad (5)$$
$$s; (\textbf{i64.const } k) \ (\textbf{i64.const } l) \ (\textbf{segment.new } o) \ \hookrightarrow_i \ \textbf{trap} \qquad \text{otherwise} \quad (6)$$
$$s; (\textbf{i64.const } k) \ (\textbf{i64.const } t) \ (\textbf{i64.const } l) \ (\textbf{segment.set\_tag } o) \ \hookrightarrow_i \ s'; \epsilon \qquad \text{if } s' = s \text{ with } \text{tag}(i, k + o, l) = t \quad (7)$$
$$s; (\textbf{i64.const } k) \ (\textbf{i64.const } t) \ (\textbf{i64.const } l) \ (\textbf{segment.set\_tag } o) \ \hookrightarrow_i \ \textbf{trap} \qquad \text{otherwise} \quad (8)$$
$$s; (\textbf{i64.const } k) \ (\textbf{i64.const } l) \ (\textbf{segment.free } o) \ \hookrightarrow_i \ s'; \epsilon \qquad \text{if } t = \text{free\_tag}(k + o) \wedge s_{\text{tag}}(i, k + o, |t|) = \text{tag}(k) \quad (9)$$
$$\wedge \ s' = s \text{ with } \text{tag}(i, k + o, l) = t$$
$$s; (\textbf{i64.const } k) \ (\textbf{i64.const } l) \ (\textbf{segment.free } o) \ \hookrightarrow_i \ \textbf{trap} \qquad \text{otherwise} \quad (10)$$
$$s; (\textbf{i64.const } k) \ \textbf{i64.pointer\_sign} \ \hookrightarrow_i \ (\textbf{i64.const } k') \qquad \text{if } k' = \text{sign}(k, k_s) \quad (11)$$
$$s; (\textbf{i64.const } k) \ \textbf{i64.pointer\_auth} \ \hookrightarrow_i \ (\textbf{i64.const } k') \qquad \text{if } k' = \text{strip}(k) \wedge k = \text{sign}(k', k_s) \quad (12)$$
$$s; (\textbf{i64.const } k) \ \textbf{i64.pointer\_auth} \ \hookrightarrow_i \ \textbf{trap} \qquad \text{otherwise} \quad (13)$$

Figure 11. Small-step reduction rules of the new instructions and added rules for load/stores. See the WASM paper [23] for the definitions of all rules and auxiliary constructs.

instruction can only be used when memory is declared. The type $tf = t_1^* \to t_2^*$ describes how the instruction manipulates the operand stack. The instruction e expects an operand stack where it pops t_1^* and pushes t_2^*.

5.2 Small-Step Reduction Rules

Figure 11 highlights how CAGE extends the WASM small-step reduction rules [23] using the notation established by previous work [44]. The lower part of Fig. 11 presents the new tag-aware load/store rules that replace the load/store rules from the WASM paper as well as new rules tailored based on the introduced instructions. To signal a trap, we reuse operators from the original WASM rules, including the trap operator. The store, s, is augmented with a storage mechanism that assigns a tag, t, to each 16-byte memory granule and a per-instance secret key k_s. The key is unique per WASM instance and ensures that leaked signatures cannot be used in another instance or in another run of the same instance. Precisely, we use the following notation:

- $t = s_{\text{tag}}(i, addr, len)$: Extracts the tag t for a memory region in instance i accessed at address $addr$ with length len, if the tag is the same for all bytes in the range $[addr, addr + len)$.
- $s' = s$ with $\text{tag}(i, addr, len) = t$: Updates the state with new tags for the memory region at address $addr$ with length len, if $addr$ is aligned to 16 bytes and the memory region is in bounds of the memory.
- $t = \text{tag}(pointer)$: Extracts the tag from a tagged pointer.
- $t' = \text{new\_tag}(t)$: Creates a tagged pointer t' from an untagged pointer t to be used for a new segment. The tag is randomly chosen from a pool of tags.

- $t' = \text{free\_tag}(t)$: Creates a tagged pointer t' for the purpose of freeing a segment. The tag is different from the tag stored in t.
- $k' = \text{sign}(k, k_s)$: Creates a cryptographic signature based on the index k and a per-instance secret key k_s and inserts it into the upper bits of k.
- $k' = \text{strip}(k)$: Removes the cryptographic signature from the upper bits of k.

Further, Figure 11 presents the added and modified components in the rules. Each reduction rule is depicted with the top of the operand stack and the state, s, on the left side, representing the pre-execution state. The resulting stack and state after the execution of the instruction is placed on the right side. For rules with $\hookrightarrow_i$, i represents the instance in which the instruction is executed. CAGE modifies the semantics of load/store instructions by adding new rules for loads and stores in Eqs. (1) to (4), which trap on tag mismatches.

Additionally, CAGE introduces new rules that express its new instructions, which are described below:

- segment.new: It creates a new, zeroed memory segment for a given pointer and size and returns a tagged pointer (Eq. (5)).
- segment.set_tag: It transfers ownership for a given pointer and size to a tagged pointer (Eq. (7)). This results in the tagged pointer being able to access the memory segment and can be used to merge adjacent segments.
- segment.free: It invalidates a segment specified by a pointer and length by tagging the segment with a new, implementation-defined tag (Eqs. (9) and (10)), ensuring accesses to the freed segment are caught. This instruction
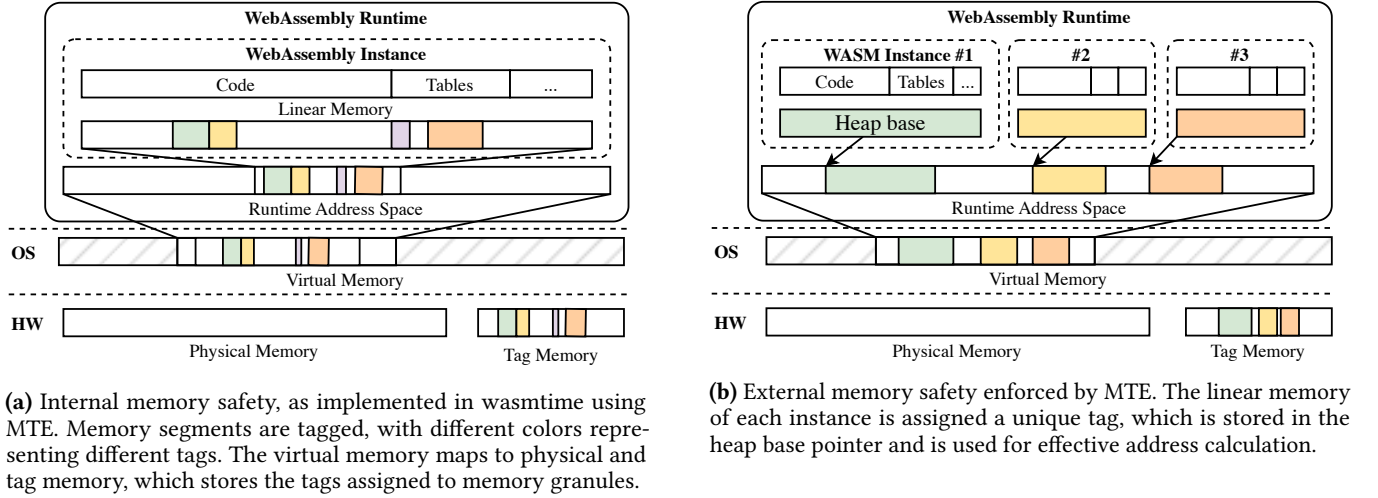
(a) Internal memory safety, as implemented in wasmtime using MTE. Memory segments are tagged, with different colors representing different tags. The virtual memory maps to physical and tag memory, which stores the tags assigned to memory granules.

(b) External memory safety enforced by MTE. The linear memory of each instance is assigned a unique tag, which is stored in the heap base pointer and is used for effective address calculation.

Figure 12. System design of internal and external memory safety as implemented using MTE in CAGE.

also traps if a segment is freed twice, i.e., the given tagged pointer cannot access the memory region.

- `i64.pointer_sign`: It signs a pointer and places a cryptographic hash in its upper bits (Eq. (11)).
- `i64.pointer_auth`: It validates a signed pointer to ensure the hash in the upper bits matches the address in the lower bits. If the hash matches, the hash is removed from the upper bits (Eq. (12)). Otherwise, it produces a trap (Eq. (13)).

Using the above rules, the Eqs. (6), (8) and (10) of Figure 11 ensure traps when code tries to create, modify, or free unaligned segments or segments outside the linear memory for a given instance. CAGE requires all segments to be aligned to 16 bytes.

6 Implementation

CAGE is built based on LLVM 17, wasi-libc, and wasmtime 16.0.0. CAGE consists of a modified LLVM toolchain (§6.1), wasi-libc (§6.2) and implementation of the memory safety extension in wasmtime (§6.3, §6.4).

6.1 LLVM Compiler Toolchain

In CAGE, we use LLVM [34] to compile our C/C++ applications to WebAssembly. We extend its existing WASM backend to be able to emit the new CAGE's instructions. We further introduce new intrinsic functions that correspond and are lowered to CAGE's WASM instructions by the compiler. The clang frontend and the CAGE compiler passes are mainly responsible for inserting calls to CAGE's new intrinsic functions when necessary. Precisely, we introduce two WASM-specific sanitizer passes that can be enabled in the LLVM pipeline via compiler flags. In clang, we introduce new built-in functions, which directly map to these intrinsics and allow programmers to use the new instructions, e.g., to build a segment-aware memory allocator.

The first sanitizer pass is designed to provide memory safety for stack allocations when compiling the source code to WebAssembly. It analyzes functions for stack allocations, applies padding, and creates the memory segments (§4.2), ensuring temporal and spatial safety of stack allocations.

The second sanitizer pass enforces pointer authentication for indirect function calls. CAGE instruments code taking references to functions and performing indirect calls. We do not handle other operations on code pointers, as pointer arithmetic on function pointers is undefined behavior in C [25, §6.5.6/2, §6.5.2/1] and C++ [26, §7.6.6/1, §6.8/8, §31.8.4/4].

Note that both sanitizer passes run after all LLVM optimizations. This ensures that CAGE does not block passes that might remove stack allocations, such as mem2reg.

6.2 WASI Libc Modifications

We port the WebAssembly System Interface (WASI) and wasi-libc to wasm64 to run applications relying on libc on wasm64. To achieve this, CAGE adapts the size and the pointer types from 32 to 64 bits. Further, we modify dlmalloc, the default allocator in wasi-libc, to provide memory safety for heap allocations. CAGE' allocator creates memory segments and returns tagged pointers to these segments. Thus, it protects both allocator metadata and adjacent allocations from being accessed or modified through heap overflows, as illustrated in Fig. 8a. When freeing or reallocating memory, segments are freed, ensuring temporal safety. Lastly, we recompile WASI-libc with pointer authentication to ensure function pointers are signed and authenticated by library code.

6.3 Internal Memory Safety

Figure 12a illustrates an overview of CAGE's implementation of the internal memory safety (§4.1) extension in wasmtime using MTE. We modify wasmtime and its supporting libraries to parse, process, and enforce the memory safety extension
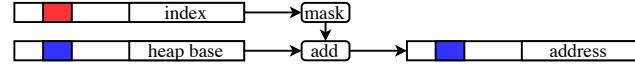
(a) Sandboxing with MTE, with all tags being reserved for the runtime to isolate WASM guests.
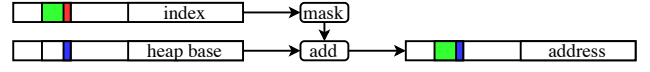
(b) MTE sandboxing and the memory safety extension combined, with three tags reserved for the guest.

Figure 13. Pointer masking to ensure bounds tags cannot be manipulated.

described in §4.2. More specifically, we add support for MTE and PAC in the form of new instructions and lowering rules. We implement segments using MTE. CAGE provides memory safety for the heap via its modified allocator that protects heap allocations by creating segments. Respectively, to provide memory safety for the stack, CAGE adapts the stack allocations; the first segment in each function is assigned a random tag and successive allocations increment the tag by one (§4.2), eventually wrapping around. This ensures adjacent allocations on the stack never share a tag. Importantly, CAGE uses synchronous MTE to trap on memory safety violations before their effects become observable.

For pointer authentication, CAGE ensures that each WASM instance receives a distinct secret key to sign pointers. If multiple WASM instances run in a single process, CAGE initializes a global random value per instance used as the PAC's instructions modifier. This is required, as PAC keys are shared per process. Otherwise, CAGE uses PAC instructions without a modifier.

6.4 External Memory Safety

Fig. 12b presents how CAGE utilizes memory tagging to replace software-based bounds checks and preserve external memory safety (§4.1) for each WASM instance. Initially, the runtime assigns a tag to each instance on module instantiation, which is stored in the heap base address. The effective memory access address is calculated by adding the accessed index to the tagged heap base address. The memory of the runtime is tagged with zero. It enables MTE to catch any access outside the sandbox due to the tag mismatch.

CAGE can further combine both its *external* and its *internal* memory safety extensions by splitting the pointer tag bits among them. The upper bits are used for internal memory safety, while the lower bits are reserved for sandboxing. In CAGE's prototype, we assign three bits for internal memory safety and one bit for enforcing external memory safety. This isolates a single WASM instance while assigning three bits for internal memory safety.

Lastly, CAGE must ensure that adding an untrusted pointer to the heap base does not allow WASM code to craft arbitrary values to escape the sandbox. To provide these guarantees, CAGE masks the WASM index before address computation, as shown in Fig. 13. CAGE masks bits 56–59 when only external memory safety is enabled (Fig. 13a) and bit 56 when both internal and external memory safety are enabled (Fig. 13b). **Limitations.** A natural limitation of CAGE is the number of sandboxes within a single process. MTE provides a limited

Table 3. Runtime benchmarking configurations.

Variant	Ptr width	Internal	External	Ptr auth
baseline wasm32	32-bit	No	No	No
baseline wasm64	64-bit	No	No	No
CAGE-mem-safety	64-bit	Yes	No	No
CAGE-ptr-auth	64-bit	No	No	Yes
CAGE-sandboxing	64-bit	No	Yes	No
CAGE	64-bit	Yes	Yes	Yes

number of tags (16). CAGE reserves the zero tag for the runtime, leaving the remaining 15 for the sandboxes. In a future version of CAGE, the number of available sandboxes can be increased by ensuring tags are only reused for memory regions outside the address range reachable by WASM pointers, i.e., via combining guard pages with memory tagging.

We must exclude certain tags from being generated by MTE instructions, as CAGE uses the tag bit 56 to distinguish between runtime and guest memory. On top of that, CAGE must also exclude the zero tag from the guest, as this tag is reserved for guard slots and untagged segments. To this end, at runtime startup, we specify which tags can be generated using the `prctl` mechanism.

7 Evaluation

7.1 Experimental Setup

We conduct our benchmarks on a Google Pixel 8 equipped with an ArmV9 Tensor G3 chip, including one Cortex-X3 (2.91 GHz, out-of-order), four Cortex-A715 (2.37 GHz, out-of-order), and four Cortex-A510 (1.7 GHz, in-order) cores. All cores feature MTE and PAC with FEAT_FPAC enabled, making PAC trap on authentication.

Methodology. We evaluate CAGE on the PolyBench/C 3.2 suite [45] and microbenchmarks with the configurations outlined in Table 3. These include wasm32 and wasm64 as baselines and the memory-safety, pointer-authentication (§6.3), and sandboxing (§6.4) components from CAGE, as well as all components of CAGE combined. We perform and compare each benchmark on every type of CPU core available on the Tensor G3 by pinning benchmarks to a single core.

7.2 Performance Overheads

Figure 14 illustrates the mean runtime overheads for the PolyBench/C benchmarks.

CAGE memory safety. Compared to wasm64, our memory safety extension has a mean overhead of 3.6±6.9 %, 5.6±4.3 %, and 1.5 ± 3.3 %. **CAGE sandboxing.** MTE-based sandboxing achieves a mean speedup of 3.7 ± 6.5 %, 5.1 ± 4.0 %, and 33.9 ± 2.4 % over wasm64. **CAGE.** When combining our memory
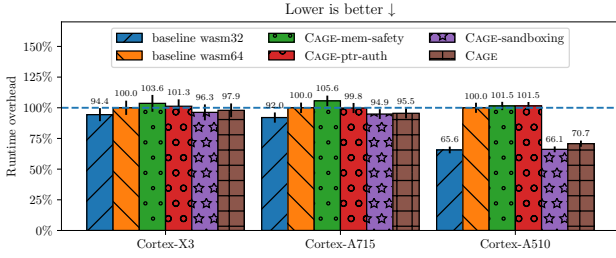
Figure 14. PolyBench/C runtime overheads of different configurations described in Table 3, normalized to wasm64.
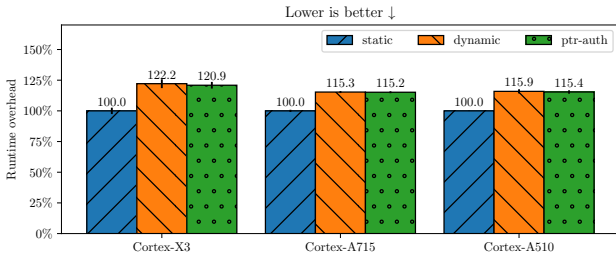

Figure 15. Overheads of pointer authentication.

safety extension with MTE-based sandboxing, we see a mean speedup of $2.1 \pm 5.6\%$, $4.5 \pm 4.1\%$, and $29.2 \pm 2.5\%$ compared to wasm64, while providing stronger security guarantees.

CAGE pointer authentication. As the PolyBench/C test suite does not exercise virtual function calls, we only see an overhead within error margins over 64-bit WASM. We perform a microbenchmark, comparing static against dynamic and authenticated dynamic function calls in Fig. 15. We observe virtually no overhead occurring as the result of pointer authentication, as adding pointer authentication only adds 5 cycles of latency, which is not noticeable. Switching from static to dynamic function calls results in a 15 %–22 % overhead, depending on the CPU being run on.

CAGE Startup Overhead. We measure the startup overhead for a WASM instance with a static memory of 128 MiB, calling a function that immediately returns. The overhead of tagging the linear memory is hidden by the runtimes startup overhead.

7.3 Memory Overheads

The memory overheads of CAGE consist of two factors: *(i)* the overhead of switching from 32-bit to 64-bit WASM and *(ii)* the overhead of memory tags caused by MTE, which is 1/32 of the memory with MTE enabled. Tags are stored in a separate physical address space, the *tag PA space* [10]. This space is not managed by the operating system and is not included in maximum resident set size (rss) measurements. We thus estimate the memory overhead with MTE by factoring in 4 bits for every 16 byte of memory, which results in an additional $1/32 = 3.125\%$. As our maximum rss measurements include the runtimes memory, which does not use MTE, this

Table 4. Variants for initializing and tagging memory.

Variant	Instr	Granule	Sets 0	memset
memset	-	-	No	Yes
stg	`stg`	16 bytes	No	No
st2g	`st2g`	32 bytes	No	No
stgp	`stgp`	16 bytes	Yes	No
stzg	`stzg`	16 bytes	Yes	No
st2zg	`st2zg`	32 bytes	Yes	No
stg+memset	`stg`	16 bytes	Yes	Yes
st2g+memset	`st2g`	32 bytes	Yes	Yes

should be considered an over-approximation; in reality, the memory overhead of CAGE is lower. We measure the mean overhead of wasm64 over wasm32 at 0.6 % and thus estimate CAGE's overhead at < 5.3 %.

7.4 Security Guarantees

We evaluate the security guarantees from the perspective of internal and external memory safety (Section 4.1).

External memory safety. CAGE implements sandboxing with MTE in synchronous mode, which enforces sandboxes through hardware. This prevents sandbox escapes, such as CVE2023-26489, from accessing memory outside the linear memory. We prevent malicious code from forging tags to escape the sandbox, as described in §6.4. However, we limit the number of sandboxes in one process to at most 15, which is required to assign a distinct tag to each sandbox and one to the runtime.

Internal memory safety. Our choice to utilize a memory-tagging-based approach does not provide complete memory safety, as we rely on a limited number of tags. As we reserve one tag for guard allocations and untagged segments, the probability for a tag collision is $1/15 = 6.\overline{6}\%$. If we utilize MTE for sandboxing, the chance of a tag collision rises to $1/7 \approx 14.3\%$. CAGE deterministically protects against off-by-one overflows, use-after-free, and double-free errors, which are caught at least until the reuse of a memory allocation. Signing function pointers further reduces the possible attack surface. As WASM already provides strong guarantees against control-flow-based attacks, adding pointer authentication primitives does not substantially improve security. CAGE protects against reusing leaked function pointers between instances and statically deducing function pointers. We recommend CAGE to be used as a secondary defense mechanism to mitigate classes described in Table 2. With the ability to deploy the prototype in production, bugs may be found in production environments not discovered by testing workloads.

Initializing tagged memory. We benchmark several primitives to initialize an uncached 128 MiB large memory region while setting its allocation tag, representing the scenario of setting up a WASM instance. We run the configurations in Table 4 with synchronous MTE and measure the results in Fig. 16. `stzg`, `stz2g`, and `stgp` are slightly faster than a raw memset, even though they initialize memory and set tags.
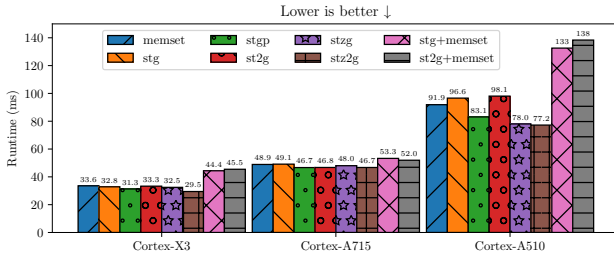
Figure 16. Performance results of the benchmarking variants from Table 4 on 128 MiB of memory.

We believe this is the case as they do not perform tag checks before accessing memory [10]. The Cortex-X3 core compensates for its lower throughput (Table 1) with a higher clock speed (2.91 GHz) compared to the Cortex-A715 (2.37 GHz).

8 Related Work

Prior research exists on enriching WASM with memory safety properties. A significant project is *MS-WASM* [16, 37], a memory safety extension for WASM that introduces a new *segment memory* distinct from the linear memory, preventing access through arbitrary offsets. The segment memory relies on accesses through unforgeable *handles*, akin to CHERI pointers [57]. On the contrary, CAGE does not introduce a distinct memory region and a new pointer type, but rather builds upon 64-bit WASM without altering its memory layout to be able to minimize its runtime overheads on devices supporting Arm's MTE.

RichWasm is another approach towards memory safety for WASM [20]. It provides a richly typed intermediate language for safe memory interactions between languages with varying memory management models. RichWasm allows for static detection of memory safety violations in the context of mixed-language interoperability between strongly typed languages like Rust or OCaml. Unsafe languages (e.g., C/C++) lack information for static safety analysis and are not directly supported by RichWasm's type-driven safety model. In contrast, CAGE targets languages that do not encode safety in their runtime or type system.

Wasm memcheck (wmemcheck) [8] is a tool in wasmtime providing the ability to check for invalid mallocs, double-frees, reads, and writes inside a WASM module, assuming certain properties about `malloc` and `free`. It is conceptually similar to Valgrind memcheck [41].

9 Conclusion

In this paper, we present CAGE, a hardware-accelerated WebAssembly toolchain to provide memory safety, consisting of a minimally invasive and adaptable WASM extension, a compiler toolchain based on LLVM, a modified wasi-libc that includes a custom allocator to provide spatial and temporal memory safety, and a wasmtime implementation, that is responsible for compiling and running the CAGE's WASM

extension using MTE and PAC. Further, CAGE improves the performance of WASM sandboxing mechanism by utilizing MTE as a replacement for software-based bounds checks. Our evaluation of CAGE highlights that CAGE is a memory safety solution for WASM that is suitable for production deployment as it incurs minimal runtime and memory overheads while providing efficient memory safety guarantees.

Software artifact. CAGE is publicly available with its entire experimental setup [19]. Detailed information can be found in Appendix A.3.

Acknowledgement

A Artifact Appendix

A.1 Abstract

This appendix provides the necessary information to build the artifacts and reproduce the experiments of the CGO'25 paper "CAGE: Hardware-Accelerated Safe WebAssembly" by M. Fink, D. Stavrakakis, D. Sprokholt, S. Chakraborty, J.-E. Ekberg., and P. Bhatotia. CAGE provides a memory safety abstraction for WebAssembly, with an implementation in LLVM to transparently compile unmodified C/C++ programs, a modified libc that provides memory safety for heap allocations, and an implementation in wasmtime that utilizes Arm MTE and PAC to implement the abstraction. We provide an artifact including pre-compiled binaries and all sources and scripts to build and evaluate the artifact to reproduce the results and figures in this paper.

A.2 Artifact Check-List (Meta-Information)

- **Program:** LLVM with CAGE modifications, source included; wasmtime with CAGE modifications, source included; wasi-libc with CAGE modifications, source included.
- **Compilation:** Clang 17, rustc 1.80
- **Transformations:** Stack allocation hardening as an LLVM pass.
- **Binary:** Pre-built binaries for wasmtime, LLVM, wasi-sdk included. Source code and makefiles to re-generate binaries included.
- **Run-time environment:** Provided binaries built for Linux 6.8.12 (nixOS) and Android 14.
- **Hardware:** We require an AArch64 device with both PAC and MTE (Pixel 8) and an x86-64 machine for cross-compilation.
- **Metrics:** Average runtime overhead; estimated memory overhead

- **Output:** PDFs for the plots; Text files with raw data for tables and claims in text.
- **Experiments:** Makefile to run all experiments is included.
- **How much disk space required (approximately)?:** 25 GiB.
- **How much time is needed to prepare workflow (approximately)?:** 2–3 hours.
- **How much time is needed to complete experiments (approximately)?:** 2–3 days.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache License with LLVM Exceptions (LLVM, wasi-libc); Apache License (wasmtime).
- **Workflow framework used?:** Makefiles.
- **Archived (provide DOI)?:** 10.5281/zenodo.13772996

A.3 Description

A.3.1 How Delivered. All source code can be found at the git repositories below, as well as the following persisted DOI for the artifact: 10.5281/zenodo.13772996, which contains all source code as well as the scripts to build and evaluate all artifacts.

- https://github.com/TUM-DSE/llvm-memsafe-wasm
- https://github.com/TUM-DSE/wasmtime-mte
- https://github.com/TUM-DSE/wasm-tools-mte
- https://github.com/martin-fink/wasi-libc

A.3.2 Hardware Dependencies. The evaluation is performed on a Google Pixel 8 with Arm MTE and PAC. We cross-compile LLVM, wasmtime, and the benchmarks on an x86 machine.

A.3.3 Software Dependencies. We require the following set of software on the Google Pixel 8 device:

- Termux
- sshd
- bash

We require the following set of software on the x86 machine to compile LLVM, wasmtime, and evaluate the benchmarks:

- Linux (tested with 6.8.12)
- nix (tested with 2.18.5): All other dependencies are fetched and pinned to a specific version using the nix package manager and can be found in the `nix/` directory in the artifact.

A.3.4 Benchmarks. We run the PolyBench/C benchmark suite to measure the runtime and memory overhead of Cage's components compared to the 32- and 64-bit baselines. To measure startup overheads, we measure the overhead of instantiating a module declaring a 128 MiB memory and calling an empty function. To measure pointer authentication overheads, we measure a modified version of PolyBench/C's 2mm benchmark, where the matrix multiplication is moved into a function call that is either performed statically or dynamically through a vtable.

A.4 Installation

To get started, download the artifact from Zenodo, navigate to the artifact directory, and run the following command to download all required dependencies using nix.

```
1 curl -L -o cgo-artifact.zip https://zenodo.org/records
      /13772996/files/cgo-artifact.zip?download=1
2 nix-shell -p unzip --run 'unzip cgo-artifact.zip'
3 cd cgo-artifact
4 cd nix
```

```
5 nix develop
6 cd ..
```

This opens a new shell with all dependencies required to build, run, and evaluate all benchmarks.

A.4.1 SSH Connection to the Pixel 8. Install Termux from the Play Store or F-Droid. Once Termux is opened, install and start sshd, then connect to the x86 machine and open a port forwarding, allowing the x86 machine to connect to the Pixel 8.

```
1 pkg install sshd
2 sshd
3 ssh -R 8023:localhost:8022 user@x86machine
```

To connect to the Pixel 8, replace the following two lines in `config.mk` with the values corresponding to your device.

```
1 export SSH_HOST=u0_a265@localhost
2 export SSH_PORT=8023
```

A.5 Experiment Workflow

All required software (LLVM, wasmtime, benchmarks) is cross-compiled on the x86 machine and copied to the Pixel 8 using a set of provided scripts.

Building LLVM, wasmtime, wasi-sdk, and the Benchmarks: *[1 human-minute + 2–3 compute-hours]*

For convenience and to reduce build times, we have included pre-built versions of wasi-sdk and LLVM in the artifact. To build them from scratch, delete the following files:

```
1 rm -rf toolchain/wasi-sdk-20
2 rm -rf toolchain/wasi-sdk-20+memory64
3 rm -rf toolchain/wasi-sdk-20+memory64+memsafety
4 rm -rf toolchain/wasi-sdk-20+memory64+memsafety+ptr-auth
5 rm -rf toolchain/wasi-sdk-20+memory64+ptr-auth
```

To build the toolchain and benchmarks, run:

```
1 make -j$(nproc) build
```

This produces the following artifacts, as well as the benchmarks used in this paper:

```
1 # wasmtime:
2 ./toolchain/wasmtime/target/aarch64-linux-android/release/
      wasmtime
3 # llvm:
4 ./toolchain/wasi-sdk-20+memory64+memsafety+ptr-auth/wasi-sdk
      -wasi-sdk-20+memory64+memsafety+ptr-auth/bin/clang
5 # wasi-sdk with different configurations
6 ./toolchain/wasi-sdk-20
7 ./toolchain/wasi-sdk-20+memory64
8 ./toolchain/wasi-sdk-20+memory64+memsafety
9 ./toolchain/wasi-sdk-20+memory64+ptr-auth
10 ./toolchain/wasi-sdk-20+memory64+memsafety+ptr-auth
```

A.6 Evaluation and Expected Result

Expected duration: *[1 human-minute + 2–3 compute-days]*
Running the experiments on the Pixel 8 devices takes a long time. This is primarily caused by our choice to run all experiments on all three types of cores found in the Pixels chipset. Running the benchmarks on the low-power Cortex-A510 cores takes up most of the runtime. To perform the evaluation, run:

```
1 make -j$(nproc) evaluate
```

This copies all benchmarks and artifacts to the connected Pixel 8, performs benchmarks, copies the results to the x86 machine, and creates plots and results claimed in the paper. The results can be found in the `results/` directory. We reproduce the following figures/claims:

- Runtime overhead (Fig. 14): `results/runtime.pdf`
- Pointer auth. overhead (Fig. 15): `results/ptr-auth.pdf`
- Memory overhead (Section 7.3): `results/mem.txt`
- Startup overhead (Section 7.2): `results/startup.txt`
- Memory tagging overheads (Fig. 16): `results/stg.pdf`

Additionally, we reproduce the following architectural analysis results from §2:

- MTE sync/async mode overhead (Fig. 4): `results/mte-mode.pdf`
- MTE instruction latencies/throughput (Table 1): `results/inst-cycles.txt`

A.7 Notes

We provide hardware with the required software preinstalled for the CGO'25 artifact reviewers.

A.8 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

References

[1] [n. d.]. Docker docs - Wasm workloads (Beta). https://docs.docker.com/desktop/wasm/. https://docs.docker.com/desktop/wasm/ Accessed on May 15, 2024.

[2] [n. d.]. Memory Safety. https://www.chromium.org/Home/chromium-security/memory-safety/. https://www.chromium.org/Home/chromium-security/memory-safety/ Accessed on March 14, 2024.

[3] 2019. CVE-2023-26489. Available from NIST National Vulnerability Database, CVE-ID CVE-2023-26489.. https://nvd.nist.gov/vuln/detail/CVE-2023-26489

[4] 2024. WebAssembly - Security. https://webassembly.org/docs/security/. https://webassembly.org/docs/security/ Accessed on May 15, 2024.

[5] 2024. WebAssembly Use Cases. https://webassembly.org/docs/use-cases/ Accessed on March 28, 2024.

[6] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2009. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 1–40. https://doi.org/10.1145/1609956.1609960

[7] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors.. In *USENIX Security Symposium*, Vol. 10. 96. https://dl.acm.org/doi/10.5555/1855768.1855772

[8] Bytecode Alliance. 2024. Wasm memcheck. https://docs.wasmtime.dev/wmemcheck.html

[9] Bytecode Alliance. 2024. Wasmtime. https://github.com/bytecodealliance/wasmtime A fast and secure runtime for WebAssembly.

[10] ARM Ltd. [n. d.]. *Arm Architecture Reference Manual for A-profile architecture.* White Paper. https://developer.arm.com/documentation/ddi0487/latest/ Accessed: 2024-03-21.

[11] ARM Ltd. 2019. *ArmV8.5-A Memory Tagging Extension.* White Paper. https://developer.arm.com/documentation/102925/latest/ Accessed: 2023-12-14.

[12] Kartal Kaan Bozdoğan, Dimitrios Stavrakakis, Shady Issa, and Pramod Bhatotia. 2022. SafePM: A sanitizer for persistent memory. In *Proceedings of the Seventeenth European Conference on Computer Systems.* 506–524. https://doi.org/10.1145/3492321.3519574

[13] Intel Corporation. 2013. Introduction to Intel(R) Memory Protection Extensions. https://software.intel.com/en-us/Articles/introduction-to-intel-memory-protection-extensions Accessed 2024-05-09.

[14] Thurston Dang, Petros Maniatis, and David Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. *ASIACCS 2015 - Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security* (04 2015), 555–566. https://doi.org/10.1145/2714576.2714635

[15] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. 2006. SAFECode: enforcing alias analysis for weakly typed languages. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (Ottawa, Ontario, Canada). ACM, New York, NY, USA, 144–157. https://doi.org/10.1145/1133981.1133999

[16] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. 2019. Position paper: Progressive memory safety for webassembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy.* 1–8. https://doi.org/10.1145/3337167.3337171

[17] Gregory Duck and Roland Yap. 2016. Heap bounds protection with low fat pointers. 132–142. https://doi.org/10.1145/2892208.2892212

[18] Gregory J. Duck, R. Yap, and L. Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. In *Network and Distributed System Security Symposium (NDSS).* https://doi.org/10.14722/ndss.2017.23287

[19] Martin Fink, Stavrakakis Dimitrios, Sprokholt Dennis, Chakraborty Soham, Ekberg Jan-Erik, and Bhatotia Pramod. 2024. *"Cage: Hardware-Accelerated Safe WebAssembly" Artifact.* https://doi.org/10.5281/zenodo.13772996

[20] Michael Fitzgibbons, Zoe Paraskevopoulou, Noble Mushtak, Michelle Thalakottur, Jose Sulaiman Manzur, and Amal Ahmed. 2024. RichWasm: Bringing Safe, Fine-Grained, Shared-Memory Interoperability Down to WebAssembly. 8 (2024), 214:1656–214:1679. Issue PLDI. https://doi.org/10.1145/3656444

[21] WebAssembly Community Group. 2024. Memory64: 64-bit Memory Indexing for WebAssembly. https://github.com/WebAssembly/memory64 Proposal for 64-bit memory addressing in WebAssembly.

[22] WebAssembly Community Group. 2024. WebAssembly System Interface - WASI. https://github.com/WebAssembly/WASI

[23] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 185–200. https://doi.org/10.1145/3062341.3062363

[24] Niranjan Hasabnis, Ashish Misra, and R. Sekar. 2012. Light-Weight Bounds Checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California) *(CGO '12).* Association for Computing Machinery, New York, NY, USA, 135–144. https://doi.org/10.1145/2259016.2259034

[25] International Organization for Standardization 2018. *ISO/IEC 9899:2018 Programming languages – C.* International Organization for Standardization, Geneva, Switzerland. https://www.iso.org/standard/74528.html

[26] International Organization for Standardization 2020. *ISO/IEC 14882:2020 Programming languages – C++.* International Organization for Standardization, Geneva, Switzerland. https://www.iso.org/standard/79358.html

[27] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: a safe dialect of

C.. In *USENIX Annual Technical Conference, General Track.* 275–288. https://dl.acm.org/doi/10.5555/647057.713871

[28] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101. https://doi.org/10.1145/3399742

[29] Pierre Krieger. 2024. The redshirt operating system. https://github.com/tomaka/redshirt

[30] Taddeus Kroes, Koen Koning, Erik Kouwe, Herbert Bos, and Cristiano Giuffrida. 2018. Delta pointers: buffer overflow checks without the checks. 1–14. https://doi.org/10.1145/3190508.3190553

[31] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. SGXBOUNDS: Memory Safety for Shielded Execution. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 205–221. https://doi.org/10.1145/3064176.3064192

[32] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. 2018. Code-pointer integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses.* 81–116. https://dl.acm.org/doi/10.5555/2685048.2685061

[33] kwast os. 2024. Kwast: Rust operating system running WebAssembly as userspace in ring 0. https://github.com/kwast-os/kwast

[34] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86. https://doi.org/10.5555/977395.977673

[35] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything old is new again: Binary security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20).* 217–234. https://dl.acm.org/doi/10.5555/3489212.3489225

[36] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. 2019. {PAC} it up: Towards pointer integrity using {ARM} pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19).* 177–194. https://dl.acm.org/doi/10.5555/3361338.3361352

[37] Alexandra E Michael, Anitha Gollamudi, Jay Bosamiya, Evan Johnson, Aidan Denlinger, Craig Disselkoen, Conrad Watt, Bryan Parno, Marco Patrignani, Marco Vassena, et al. 2023. Mswasm: Soundly enforcing memory-safe execution of unsafe code. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 425–454. https://doi.org/10.1145/3554344

[38] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2019. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16.* Springer, 23–42. https://doi.org/10.1007/978-3-030-22038-9_2

[39] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 245–258. https://doi.org/10.1145/1543135.1542504

[40] George C Necula, Scott McPeak, and Westley Weimer. 2002. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* 128–139. https://doi.org/10.1145/1065887.1065892

[41] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices* 42, 6 (2007), 89–100. https://doi.org/10.1145/1273442.1250746

[42] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. [n. d.]. Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack. 46, 1 ([n. d.]), 111–112.

https://doi.org/10.1145/3292040.3219662

[43] Benjamin C Pierce. 2002. *Types and programming languages.* MIT press. ISBN: 9780262162098.

[44] Gordon D Plotkin. 1981. A structural approach to operational semantics. (1981).

[45] Louis-Noel Pouchet. [n. d.]. Polybench: The polyhedral benchmark suite. https://web.cs.ucla.edu/~pouchet/software/polybench/ Accessed: 2024-03-25.

[46] Qualcomm Technologies, Inc. 2017. *Pointer Authentication on ArmV8.3: Design and Analysis of the New Software Security Instructions.* White Paper. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf Accessed: 2023-12-14.

[47] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12).* 309–318. https://doi.org/10.5555/2342821.2342849

[48] Kostya Serebryany, Chris Kennelly, Mitch Phillips, Matt Denton, Marco Elver, Alexander Potapenko, Matt Morehouse, Vlad Tsyrklevich, Christian Holler, Julian Lettner, et al. 2023. GWP-ASan: Sampling-Based Detection of Memory-Safety Bugs in Production. *arXiv preprint arXiv:2311.09394* (2023). https://doi.org/10.1145/3639477.3640328

[49] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. 2018. Memory Tagging and how it improves C/C++ memory safety. *arXiv preprint arXiv:1802.09517* (2018).

[50] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy.* IEEE, 48–62. https://doi.org/10.1109/SP.2013.13

[51] Raven Szewczyk, Kimberley Stonehouse, Antonio Barbalace, and Tom Spink. 2022. Leaps and bounds: Analyzing WebAssembly's performance with a focus on bounds checking. In *2022 IEEE International Symposium on Workload Characterization (IISWC).* IEEE, 256–268. https://doi.org/10.1109/IISWC55918.2022.00030

[52] Gavin Thomas. 2019. A proactive approach to more secure code. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/ Accessed on March 14, 2024.

[53] Jeff Vander Stoep and Chong Zhang. 2019. Queue the Hardening Enhancements. https://security.googleblog.com/2019/05/queue-hardening-enhancements.html. https://security.googleblog.com/2019/05/queue-hardening-enhancements.html Accessed on March 14, 2024.

[54] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. 1993. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles.* 203–216. https://doi.org/10.1145/168619.168635

[55] Robert N M Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W Moore, Edward Napierala, Peter Sewell, and Peter G Neumann. 2020. CHERI C/C++ Programming Guide. (June 2020).

[56] Elliott Wen and Gerald Weber. 2020. Wasmachine: Bring IoT up to Speed with A WebAssembly OS. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops).* 1–4. https://doi.org/10.1109/PerComWorkshops48775.2020.9156135

[57] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 457–468. https://doi.org/10.1145/2678373.2665740