SCHOOL OF COMPUTATION,
INFORMATION AND TECHNOLOGY —
INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Hardware-Assisted Memory Safety for WebAssembly

Martin Fink

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Hardware-Assisted Memory Safety for WebAssembly

Hardwaregestützte Speichersicherheit für WebAssembly

Author:	Martin Fink
Supervisor:	Prof. Pramod Bhatotia
Advisor:	Dimitrios Stavrakakis
Submission Date:	15.04.2024

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.04.2024 Martin Fink

Acknowledgments

Abstract

Contents

1 Introduction

In recent years, WebAssembly (WASM) [3] has gained prominence as a versatile compilation target, catering not only to web-based applications but also to a broader spectrum of use cases. At its core, WASM is engineered to serve as an efficient compilation target for high-level, compiled languages such as C and C++. A fundamental aspect of its design is its linear, contiguous memory model. This model is instrumental in facilitating the straightforward compilation of these languages. Moreover, it plays a pivotal role in ensuring the performance efficiency of the execution process.

While WebAssembly provides a sandbox for the guest code, which protects the host and other guests from malicious or buggy code, it does not inherently prevent memory safety issues within an application's memory space. This limitation becomes particularly evident when compiling languages like C or C++, where there are no language-level guarantees to prevent these bugs.

However, recent advancements in hardware, such as ARM's Memory Tagging Extension (MTE) and the Pointer Authentication Extension (PAC), offer promising, high-performance solutions. These hardware extensions are designed to effectively address and rectify such memory safety concerns.

We introduce SafeWasm, a comprehensive solution that leverages both MTE and PAC to address both spatial and temporal memory safety bugs in WebAssembly-compiled programs. Our approach requires no modification to the source code. The core contributions of this paper are outlined as follows:

WebAssembly Extension: We develop a minimal and generic extension to the WebAssembly instruction set, allowing for protected memory regions without code changes.

Compiler Toolchain: We develop a specialized compiler toolchain for unmodified C/C++ programs, optimized for efficiency with our WebAssembly extension.

Wasmtime Modification: Modifications to Wasmtime enable leveraging MTE and PAC, providing security without compromising performance.

Bounds Checks with MTE: We leverage MTE to eliminate expensive bounds checks for 64 bit WebAssembly programs

Performance Evaluation: Extensive evaluation on ARM hardware, analyzing performance, memory overheads, and safety guarantees.

2 Background

In this section, we discuss Memory Safety, WebAssembly (WASM), and ARM's Pointer Authentication (PAC) and Memory Tagging Extension (MTE) extensions.

2.1 WebAssembly

WebAssembly [3], initially designed as an alternative, high-performance compilation target to Javascript, continues to be applied in a wide variety of use cases. WASM was carefully designed to allow for compilation from high-performance languages traditionally compiled to native machine code such as C, C++, or Rust, and for compilation to different native architectures.

Linear memory: WASM provides a linear memory that can be accessed by 32 or 64 bit integers. This allows the compilers and languages to manage memory themselves, without being forced into an unnatural idiom. Languages may ship their own allocators, garbage collectors, and lay out data structures as is efficient. On the host, this linear memory can then be mapped directly to the host memory.

Structured control flow: In WASM, unstructured control flow is not possible. Indirect function calls are made through type-checked tables, with function pointers being represented as indices into them. Jumps are realized using a set of well-defined control flow constructs. This not only reduces the attack surface of programs compiled to WASM, but also aides code generation. TODO: [explain why]

Stack machine: Since compilation targets offer different sets of registers, WASM does not expose registers, but instead operates on a typed stack. The stack can be verified to be well-typed and compiled to register machines or intermediate representations (IRs) of optimizing compilers in a single pass, without making assumptions about the number of available registers. These can vary not only between different architectures, but also between runtimes, as they might reserve some registers for their own use (such as dedicating a register to hold some global state).

Limited datatypes: WASM defines four basic data types: 32 and 64 bit integer and floating-point types respectively. Different proposals add vector-, garbage-collected-,

> and reference types, which cover other use cases. There is no distinction between pointer- and integer types, however. While this drops information, such as pointer provenance, from the original program, and prevents optimizations, this is not considered a problem. WASM is intended to be generated by an optimizing compiler, which should perform optimizations that require pointer provenance.

Since its inception, WebAssembly has expanded its utility beyond the initial design goal to various other domains, such as Function as a Service (FaaS) workloads, as an alternative to containers within Docker, as an isolation mechanism to allow untrusted code to be run within native applications, among other uses.

2.1.1 WebAssembly Sandbox

When accessing memory, the WASM runtime must ensure the access is within the bounds of the accessible linear memory. Then, the memory access is performed relative to the memory's base address. In current runtimes, this is usually achieved using two major approaches.

1. Explicit bounds checks: An explicit bounds check is inserted before each memory access, comparing the index with the bound of the current memory.

2. Guard pages: When running 32 bit WASM on 64 bit hosts, the runtime can leverage the fact that virtual memory is abundant. For each linear memory, 2^{32} bytes, or 4 GiB of virtual memory are allocated, with the memory beyond the guard being marked as inaccessible. The MMU will catch accesses into these pages and the operating system will deliver a segmentation fault to the runtime, which in turn will deliver a trap to the WASM program.

While the design of WebAssembly is designed to prevent malicious or erronous programs from compromising the host, buggy programs are still vulnerable to classical memory safety errors discussed earlier, such as buffer overflows.

2.2 Memory Safety in the context of WebAssembly

Programs written in languages like C or C++ are prone to memory safety bugs such as memory accesses to out-of-bounds or dangling pointers, which are the fundamental attack primitives enabling a whole class of attacks on a vulnerable or buggy program [9]. Approaches to tackle these issues exist in several forms. Programs may be written in managed languages that prevent these attacks by either not providing raw pointer accesses, such as Java, Python, JavaScript, or others. In these languages, access to memory is either performed through bounds-checked arrays or through managed objects, such as classes. The runtime cleans up dangling objects through a garbage collector, they cannot be manually deallocated. This results in all references pointing to valid objects and all memory accesses being bounds-checked. Other languages, such as Rust, take a different approach. In safe Rust, the type system forbids many cases of invalid programs, e.g., dangling references or raw pointer accesses. An escape hatch in the form of unsafe exists, which allows dropping down to the level of C, being able to directly manipulate raw bytes. Both these approaches represent a fundamental tradeoff. Managed memory, either in the form of reference-counting, or through a garbage collector incurs an overhead that may or may not be tolerated in some environments.

Studies show that in large software projects, memory safety bugs make up between 70 and 75% of all issues [5, 10, 11].

Lehmann et al. show that while some attack surfaces, such as jumping to arbitrary addresses or injecting shellcode, are mitigated by the design of WebAssembly, buffer overflows or dangling pointer accesses are still possible [4]. Since WebAssembly does not provide separate read-only memory regions, this opens up other surfaces, allowing attackers to potentially overwrite static data, since compilers place these in readable and writeable memory.

2.2.1 Software-Based Mitigations

To mitigate memory safety bugs in languages that do not provide safety guarantees at the language level, software-based approaches to detect these errors exist. Checks can be inserted automatically at the compiler level, which is an approach chosen by Address Sanitizer (ASAN) [7]. ASANincurs a large overhead, on average 73%, which is too large to be deployed in production and is usually only tolerated while testing or fuzzing software. A sampling based version of ASAN, GWP-Asan, is deployed in production in several large projects, which results in a low overhead, but no full protection for a single process [8]. On a large scale, however, this approach allows for discovery of real-world bugs that may not be triggered by testing or fuzzing workloads.

2.3 Memory Safety Hardware Extensions

As an alternative to flexible, but slow software solutions to detect and prevent memory safety issues, CPU designers have come up with several hardware extensions designed as a building block. These provide important primitives, that can be used to provide full or partial memory safety to programs, while also having a small enough footprint to be able to ship these solutions in production.

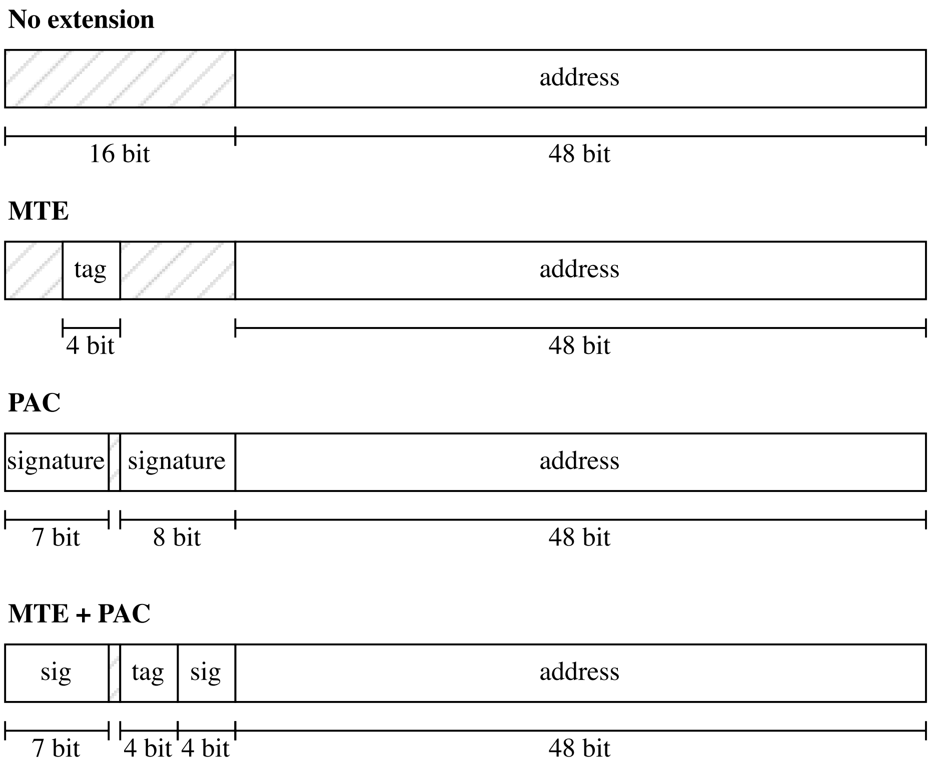

Figure 2.1: Pointer layout on aarch64 with and without MTE and PAC enabled

In fig. 2.1, the layout of pointer bits used for address translation in aarch64, the 64 bit variant of the ArmV8 instruction set, is shown. Only 48 out of the available 64 bits are used to address memory, while the remaining bits are set to either 0 or 1 to differentiate between kernel and userspace addresses, but unused for further address translation. Hardware extensions such as MTE (section 2.3.1) and PAC (section 2.3.2) utilize those unused bits to store metadata.

2.3.1 Memory Tagging Extension (MTE)

ARMs Memory Tagging Extension, available from armv8.5, provides a building block to prevent both spatial and temporal memory safety violations [1]. MTE implements a lock-and-key mechanism where memory regions can be tagged with one of 16 distinct tags, and memory accessed are only allowed using pointers with the corresponding keys.

The locking mechanism is implemented by storing a 4 bit tag in bits 56-59. Accordingly, a tag is assigned to every 16 bytes of tagged memory. When accessing memory with a mismatched tag, depending on the selected configuration, either a synchronous or asynchronous exception is thrown by the hardware. Synchronous exceptions trigger a signal to the application immediately, and the memory access does not succeed. Asynchronous exceptions are not triggered immediately and the memory access succeeds. On the next entry to the kernel, the signal is delivered to the application.

2.3.2 Pointer Authentication (PAC)

Pointer Authentication (PAC) [6] introduces primitives to prevent attackers from modifying pointers stored in memory. The extension provides three instructions that can be used to sign pointers, and sign or strip the signature from pointers. The signature is stored in the upper 16 bits of pointers, with the exact layout dependent on the Operating System, hardware, and other factors, such as if MTE is enabled. Signed pointers are invalid and cannot be used to address memory. They are created using the `pac*` instructions. Before being used, the signature needs to be removed. This happens either with the `aut*` instructions, which remove the signature if it is valid, or produce a pointer that will trap when used, if the signature does not match the address. To remove the signature regardless if it is valid, the extension provides `strip` instructions.

Both 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.

3 Motivation

While WebAssembly provides strong safety and security guarantees, as discussed in section 2.1, these mostly guarantee safety for the host running the programs, not the program itself. In [4], Lehmann et al. show that while some attack surfaces, such as injecting shellcode or jumping to arbitrary addresses, are mitigated by the design of WebAssembly, others, such as buffer overflows or write accesses to static, read-only data is possible and being used to exploit programs running in the wild. These need to be either mitigated at the language level by rewriting software in a safe language such as Rust, by manually inserting bounds checks, which is error-prone, or by inserting checks using the compiler and sanitizing the code.

If the input string is not terminated by a null terminator or longer than 32 bytes, the write will go out of bounds of the buffer and, depending on the stack layout chosen by the compiler, overwrite a.

Additionally, bugs like CVE-2023-4863 [2] continue to be exploited and show that memory safety is not a solved problem. While they do not escape the WebAssembly sandbox, they still pose a security risk. In C, the use of unsafe primitives or bugs, such as missing bounds checks, can be exploited by the attackers, e.g. by overwriting a variable to elevate their privileges. In fig. 3.1, the lack of bounds checks allows an attacker controlling the variable `input` to set `is_admin` to true.

To contain potentially malicious programs within their sandbox, a number of different techniques may be used, as discussed in section 2.1.1. In most cases, the sandbox relies on virtual memory and guard pages to contain memory accesses. However, in

```c
struct User {
    char name[32];
    int is_admin;
};
void func(char *input) {
    User user = { 0 };
    strcpy(user.name, input);
}

```

Figure 3.1: Vulnerable overflow

cases where virtual memory cannot be used, e.g. when running 64 bit WebAssembly programs, the engine needs to fall back to software based bounds checks, which bring along a heavy performance penalty. In our measurements, removing guard pages and performing explicit bounds checks showed an average overhead of TODO: [x] % overhead.

TODO: [insert graph here]

4 Overview

WebAssembly provides defences against a whole class of safety issues. The lack of unstructured control flows, e.g. jumps to dynamic addresses, and indirect calls through typed and checked tables prevent a whole class of memory safety issues such as return-orentied-programming (ROP) or type confusion when calling function pointers. Memory accesses are guaranteed to not escape its assigned linear memory. This is implemented either using explicit bounds checks by inserting a branch before each load/store, that compares the index being accessed with the size of the memory. As WebAssembly indices are only 32 bits wide, when running on 64 bit hosts, the WebAssembly runtime can request virtual guard pages, which will not be allocated as physical pages. These guard pages cover all memory indexable with a 32 bit index, and are marked as inaccessible. Only the accessible memory is marked as such. If a buggy or malicious WebAssembly program accesses memory outside its linear memory, the access will hit one of the guard pages, which triggers a segmentation fault. This fault is then caught by the WebAssembly runtime, which terminates the WebAssembly using a trap.

However, this approach is only possible for programs targeting wasm32. With the upcoming memory64 proposal[1], indices are expanded to 64 bits, 48 of which are used to index memory. Here, explicit bounds checks are required, as the virtual address space is too small for guard pages that cover all addressable memory. This incurs a significant overhead, as we show in section 7.5.

Memory Safety within WebAssembly programs While WebAssembly provides these security guarantees that prevent return-oriented programming (ROP) or sandbox escapes, programs compiled to WebAssembly still suffer from classical memory safety errors such as buffer overflows or use-after-frees. This is critical for complex programs processing untrusted input, such as image or video processing applications.

We present SafeWasm, an end-to-end solution to provide efficient spatial and temporal memory safety guarantees for programs compiled to WebAssembly. We propose a minimal extension to the WebAssembly instruction set, with primitives tailored to tagged memory, but still general enough so they can be implemented using different

[1] https://github.com/WebAssembly/memory64

types of hardware extensions or even in software. SafeWasm implements our proposed extension utilizing LLVM and wasmtime. We also present a technique to eliminate software-based bounds checks using MTE for wasm64 by tagging the linear memory accessible by untrusted user code with a different tag than the runtimes memory, which should not be accessed by WebAssembly code directly. This allows running wasm64 without the large overhead of explicit bounds checks on MTE hardware.

5 Design

5.1 Threat Model

In this threat model, we differentiate between two aspects of memory safety:

TODO: [insert graphics highlighting trusted/untrusted components]

- **Internal Memory Safety**: Ensures memory safety within the boundaries of a sandbox.
- **External Memory Safety**: Maintains the memory safety of the sandbox itself against potentially malicious programs.

Internal Memory Safety For internal memory safety, the program within the sandbox and its runtime are considered trusted. Untrusted input (e.g., network data, file reads) originates from outside the sandbox. This model mirrors the threat environment of a standard non-WASM program. Potential threats include:

Buffer overflows Attempts to write data beyond allocated buffer boundaries.

Use-after-free Accessing memory after it has been deallocated.

WebAssembly's design inherently mitigates certain threats common in non-WASM environments, so we will not consider the following vectors:

Return-oriented attacks WASM's structured control flow constructs prevent arbitrary code execution through stack manipulation.

Calling unknown function pointers Function tables enforce a strict mechanism for function calls, ensuring the integrity of call targets.

External Memory Safety For external memory safety, we focus on the security of the sandbox. Threats originate from running untrusted programs, which may be buggy or adversarial.

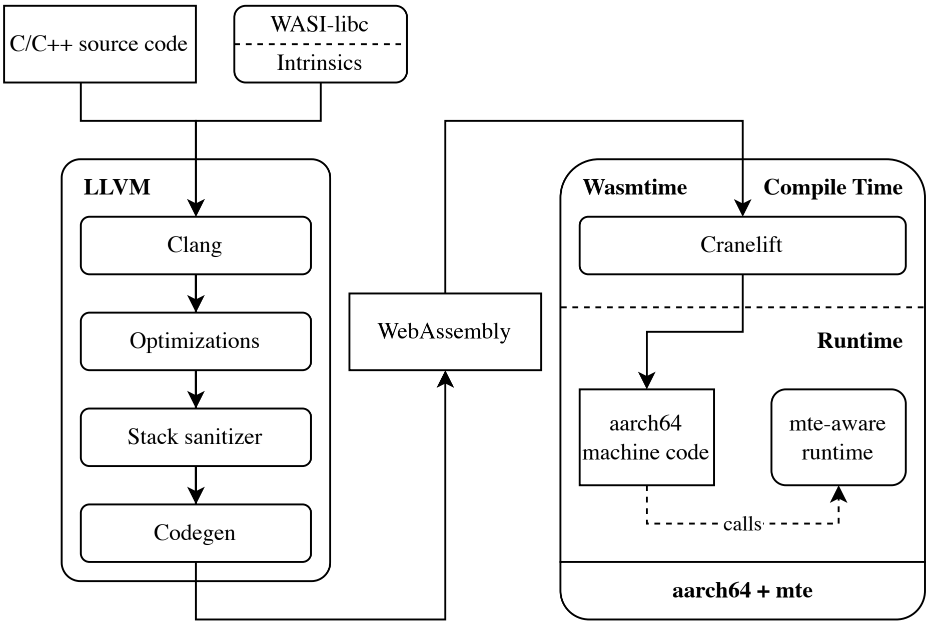

Figure 5.1: Overview

- **Sandbox escapes**: Attempts to break out of the sandbox's restrictions and access host resources.
- **Side-channel attacks**: Exploiting timing differences or resource usage patterns to infer sensitive information.

We assume that the runtime compiling and executing, as well as the operating system and underlying target architecture is free of bugs that might be exploited by malicious targets. This includes assumptions that the WASM to native compilers inserts spectre guards where appropriate and properly manages bounds checks, e.g. by inserting appropriate guard pages.

Figure 5.1 presents an overview of SafeWasm. In the WebAssembly context, a module contains one or more heaps, representing the linear memory accessible to the module. Unlike programs compiled to native binaries, WebAssembly programs access memory through indices starting at zero, with zero being a valid address. The runtime is responsible for setting up the linear memory at startup and translating indices to actual memory addresses by adding the heap base address to the index when accessing memory. Crucially, the runtime is also required to perform bounds checks for memory accesses, which account for a large runtime overhead.

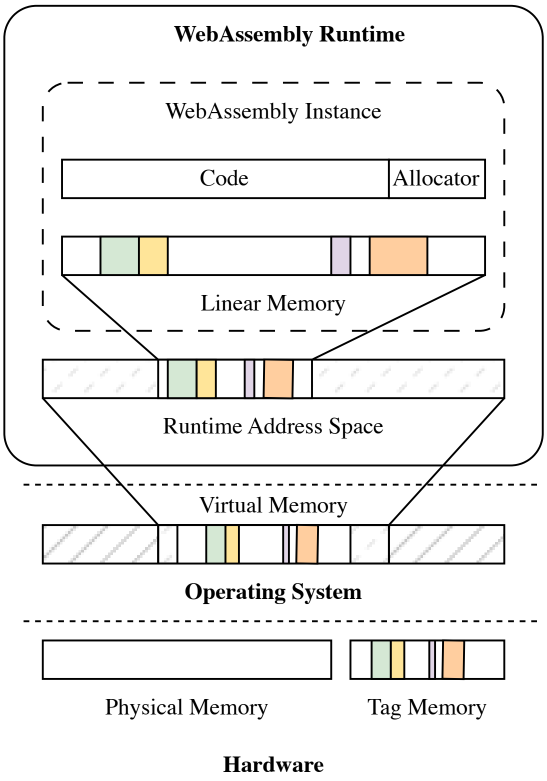

Figure 5.2: System Design of our Memory Safety Extension.

We build on top of wasm64, which allows indexing up to 2^{48} bytes, crucially leaving 16 unused bits in each address. This aligns with most hardware implementations, which also just use 48 bits of an address to index into memory.

We reserve these bits to store metadata for each pointer. Since WebAssembly does not have a dedicated pointer type, but uses plain integer types as pointers, these bytes can be manipulated using integer instructions. We do, however, introduce three primitives in the form of new instructions to create and manipulate tagged memory regions, which also set the metadata tags in pointers.

- `segment.new(i64, i64) → i64`: Allocates a new memory segment at the specified index and size, assigning it a randomly generated tag.
- `segment.set_tag(i64, i64, i64)`: Applies a user-defined tag to the memory segment identified by the given index and size.
- `segment.free(i64, i64)`: Deallocates a memory segment, simultaneously re-

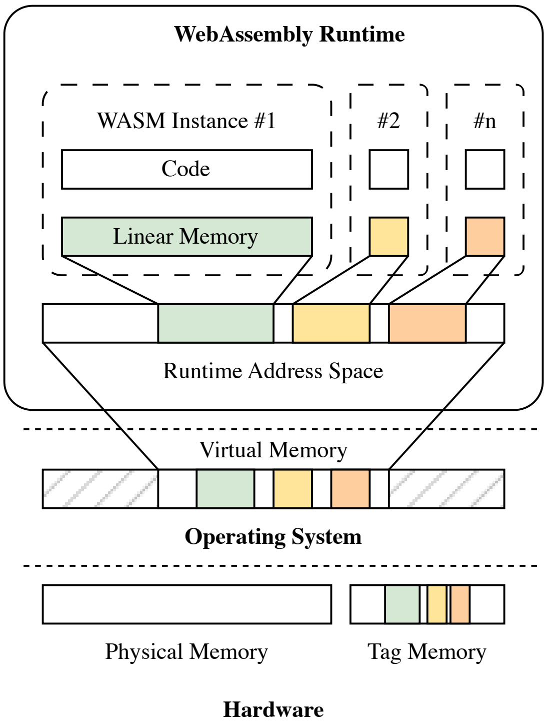

Figure 5.3: Bounds checks are replaced with MTE

tagging the corresponding index and size range to invalidate any pointers referencing this area.

The runtime environment raises a trap for any instruction that accesses memory segments with non-matching tags.

Existing instructions that access memory are inherently compatible with this system and do not require any modifications. They are capable of handling both tagged and untagged indices. This design choice allows the gradual integration of safety primitives into specific parts of WebAssembly applications where enhanced security is required. For instance, it enables the introduction of a customized malloc implementation, which prevents spatial and temporal safety bugs for heap-allocated memory.

TODO: [write about alignment]

The following C code will be used to demonstrate how the generated code will look like.

```c
char buf[64];
// ...
return;
```

```
;; Allocate space on the stack
global.get $__stack_pointer
i64.const 64
i64.sub
global.tee $__stack_pointer

;; tag the segment
i64.const 64
segment.new
local.set $buf

;; ...

;; retag with stack pointer tag
local.get $buf
global.get $__stack_pointer
i64.const 64
segment.set_tag

;; reset stack pointer
global.get $__stack_pointer
i64.const 64
i64.sub
global.set $__stack_pointer
```

5.2 Heap Safety

Heap safety is solved by aligning all allocations to 16 bytes and tagging the memory before returning to the caller. When freeing memory, the freed memory is retagged to detect use-after-free errors.

5.3 Stack Safety

To achieve stack safety, we also tag stack allocations when entering a function and retag them when leaving a function. However, tagging each stack slot is impractical due to significant performance and memory overheads. The need to align each stack slot to the tag granularity adds to this issue, leading to excessive stack space consumption and elevated performance costs, particularly for stack slots that do not require this protection, such as slots for scalar variables.

To address this challenge, we propose an algorithm that identifies memory regions within the stack that do not require protection, thus avoiding tagging slots that do not contribute to stack safety. Our approach focuses on tagging only those stack slots that are indexed using indices that cannot be statically verified to remain within the slot's size. Our analysis is not interprocedural to keep the algorithm simple. Slots that have their address taken, thus escape our analysis and are also tagged. Below we show a simplified version of our algorithm:

```
$allocsToInstrument \leftarrow \emptyset$
for $alloc \in allocations$ do
    if escapes($alloc$) then
        $allocsToInstrument$.push($alloc$)
    end if
    if isUsedByUnsafeGEP($alloc$) then
        $allocsToInstrument$.push($alloc$)
    end if
end for
for $alloc \in allocsToInstrument$ do
    insertTaggingCode($alloc$)
    insertUntaggingCode($alloc$)
end for
```

In the code example below, `i` is safe, as it is not accessed using indices and its address does not escape. The variables `bytes_read` and `buf` are determined to be unsafe, as their address escapes. Additionally, `buf` is accessed using an untrusted index, potentially leading to buffer overflows.

```c
char function(int index) {
  int i = 0; // safe
  int bytes_read = 0; // unsafe
  char buf[32]; // unsafe
  read_input(buf, &bytes_read);
  return buf[index];
}
```

This method effectively balances the need for stack safety with performance and memory efficiency constraints. Before returning, we ensure that all stack slots are untagged. This provides temporal safety, preventing stack slots from being accessed after returning from a function.

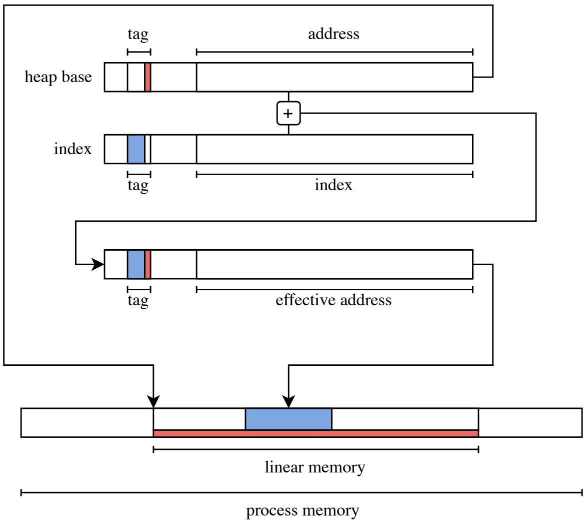

Figure 5.4: Bounds checks are replaced with MTE

5.4 Bounds Checks

WebAssembly runtimes ensure that applications remain within their allocated memory sandbox. This is traditionally achieved through bounds checks or by leveraging the operating system's virtual memory management. In wasm32, runtimes typically employ memory-mapping techniques, statically mapping the entire 32 bit addressable space and marking unpermitted areas as inaccessible. Accessing these inaccessible pages results in a segmentation fault, which is caught by the runtime, effectively containing the guest code within its sandbox.

However, this strategy faces limitations in wasm64 due to the impracticality of mapping the entire 64-bit address space. This constraint necessitates the insertion of bounds checks, which can significantly degrade performance. We demonstrate the impact of these checks on wasm64 execution speed in section 7.

In TODO: [ref:fig_design_bounds_checks], we showcase a approach that utilizes memory tagging to replace traditional bounds checks in wasm64 environments. On module instantiation, the runtime assigns a predefined tag to the linear memory and the heap base address. Memory access translation then involves adding this tagged heap base address to the accessed index.

Combining memory safety with MTE bounds checks

To combine these two approaches, it is important to not allow users to escape from the sandbox by forging tags but still benefit from the performance gained from removing bounds checks.

There is two challenges we need to take care of.

1. Adding a tagged user pointer to the heap base address should be performant and result in the cprrect tag for the respective memory.
2. It's crucial to prevent users from creating tags that enable access beyond their allocated memory sandbox.

To circumvent these issues, designate the lowest tag bit to determine whether memory belongs to the runtime or to the linear memory. Inaccessible memory us tagged with 0, while the linear memory is initially tagged with the tag 1. The remaining three tag bits are used for the memory safety features and can be generated by `segment.new`. When tagging memory using these primitives, we set the lowest tag bit to 1 before tagging. Since the memory index is untrusted, an attacker may craft a value that overflows the tag bits when adding it to the heap base, resulting in a tag that allows accessing memory outside the linear memory. To prevent this, we need to set the lowest tag bit via a bitwise mask before each memory access. The code for this can be seen in listings 5.4 and 5.4.

```
// tagging memory
v0 = or tag, 0x0100_0000_0000_0000
stzg v0, [address]
```

```
// accessing memory
v0 = add heap_base, index
v1 = or v1, 0x0100_0000_0000_0000
```

With this approach, we prevent attempts by untrusted code to access runtime memory through forged tags.

6 Implementation

Our implementation of SafeWasm is integrated into the LLVM framework, wasi-libc, and the wasmtime WebAssembly runtime. The following subsections detail the specific modifications and extensions we made to each component.

6.1 LLVM Extensions

In LLVM, we introduced a novel sanitizer pass, that can be enabled via a compiler flag, designed to provide memory safety for stack allocations when compiling to WebAssembly. This sanitizer analyzes functions for stack allocations and applies padding and tagging to them. This ensures spatial memory safety by ensuring neighbouring allocations are tagged with different tags and temporal safety by untagging the stack frame when returning from a function. As WebAssembly does not support exceptions or C-style long jumps, we do not have to handle these special cases, which have proved to be tricky to get right with tagged memory.

We extended LLVM with new built-in functions, which can be invoked directly from C code. These functions provide programmers direct access to our newly introduced memory tagging primitives, allowing for fine-grained control over memory operations. Utilizing these built-in functions, we modified the default allocator in wasi-libc to provide memory safety for heap allocations.

6.2 WASI Libc Modifications

To allow us to run applications relying on libc on wasm64, we had to port the WebAssembly System Interface (WASI) and wasi-libc to wasm64. This mainly consisted of mechanical work, changing size and pointer types to 64 bits.

To provide memory safety for heap allocations, we modified dlmalloc, the default allocator in wasi-libc. Before returning allocated memory to the user, we tag the memory. This has the additional effect of not only protecting adjacent allocations from being accessed via memory overflows, but also the allocator metadata itself. When memory is freed or reallocated, we untag the memory regions returned to the allocator.

This provides temporal safety, even after freed memory is allocated again, as the tag for the new allocation will be chosen at random.

6.3 WebAssembly Runtime Enhancements

Our implementation in the wasmtime runtime focuses on combining performance with security. In environments where hardware supports MTE, our new instructions are lowered to MTE instructions, leveraging the hardware's capabilities for memory safety. We have implemented a number of optimizations to ensure our generated code runs efficiently. In the general case, we generate a loop over the memory region to be tagged. This loop, however, is unrolled to tag larger blocks of memory at once, and is unrolled to a fixed number of instructions, if the size of the memory region to be tagged is known at compile time.

7 Evaluation

7.1 Experimental Setup

7.1.1 Testbed Configuration

We conducted our benchmarks on a Google Pixel 8 equipped with a Google Tensor G3 chip, comprising 1 × Cortex-X3 (2.91 GHz), 4 × Cortex-A715 (2.37 GHz), and 4 × Cortex-A510 (1.7 GHz) cores, with Memory Tagging Extension (MTE) enabled. To mitigate thermal throttling, we attached a cooling fan to the device. Additionally, we pinned benchmarks to the low-power Cortex-A510 cores, which significantly decreased thermal throttling and noise in our benchmarks. TODO: [should we leave this in?] As of the date of writing, this is the sole commercially available device featuring MTE.

Variant	64-bit	memsafe	bounds-chk
wasm32	No	No	No
wasm64	Yes	No	No
mem-safety	Yes	Yes	No
mte-bounds	Yes	No	Yes
all	Yes	Yes	Yes

Table 7.1: Benchmarking Variants

7.1.2 Benchmark Variants

The benchmarks utilized were from the Polybench-C suite. TODO: [add more real-world benchmarks] The variants compared are detailed in Table 7.1.

7.2 Performance Overheads

Figure 7.1 illustrates the runtime overheads for PolyBench-C benchmarks. The mean runtime compared to the baseline for the memory safety implementation is 100.8 %, with a maximum of 109.4%. With memory safety features disabled, and MTE employed for bounds checking, we reach a median runtime of 70.9% relative to the baseline, with minimum and maximum runtimes of 40.4% and 93.1%, respectively.

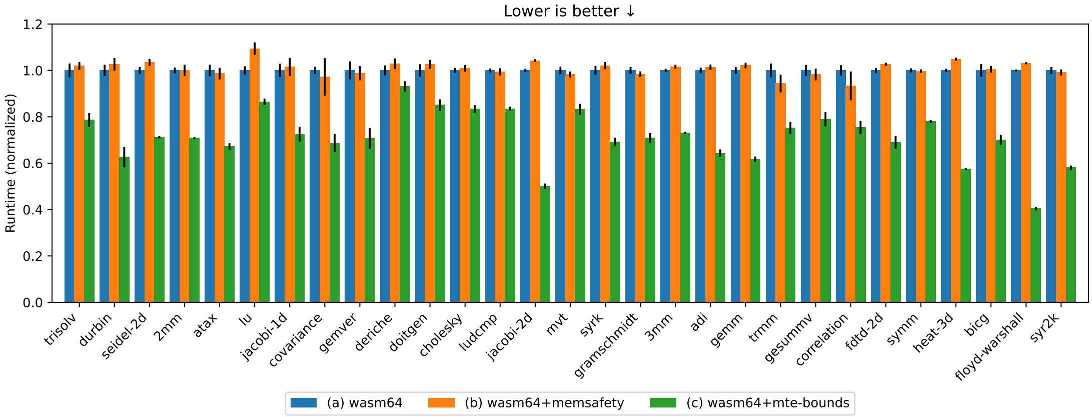

Figure 7.1: PolyBench-C runtime overheads of SafeWasm. (a) Baseline. (b) Relative overhead of wasm64 with memory safety features. (c) Bounds checks implemented using MTE.

TODO: [combining bounds + memsafe]

7.3 Memory Overheads

We measure the memory overhead with GNU time.

Memory tagging incurs overhead, particularly for small allocations due to the 16-byte alignment required for MTE. The MTE backend further introduces overhead, as observed in Figure 7.2, due to OS-level memory allocation requirements when MTE is active. Enabling our memory safety mechanism results in an average overhead of 1%, with a range of -0.67% to 23.5%. In contrast, replacing bounds checks with MTE, which necessitates tagging the entire WebAssembly linear memory, leads to an average memory overhead of 0.26%, with a range of -0.7% to 1.4%.

7.4 Security Guarantees

TODO: [insert evaluation here]

7.5 Overhead of wasm32

TODO: [insert evaluation here]

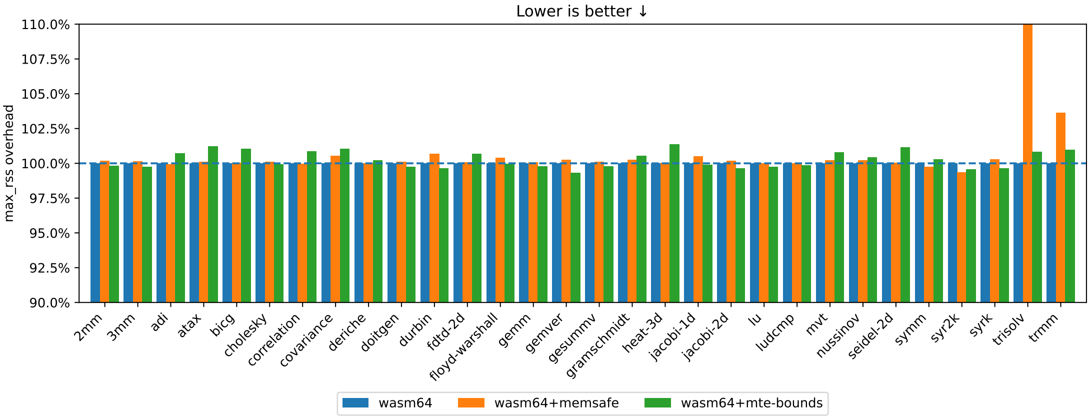

Figure 7.2: PolyBench-C memory overheads of SafeWasm. (a) Baseline. (b) Relative overhead of wasm64 with memory safety features. (c) Bounds checks implemented using MTE.

8 Related Work

9 Conclusion

Abbreviations

WASM	WebAssembly
MTE	Memory Tagging Extension
PAC	Pointer Authentication
IR	intermediate representation
ASAN	Address Sanitizer

List of Figures

List of Tables

Bibliography

[1] ARM Ltd. *Armv8.5-A Memory Tagging Extension*. White Paper. Accessed: 2023-12-14. 2019. URL: `https://developer.arm.com/documentation/102925/latest/`.

[2] *CVE-2023-4863*. Available from NIST National Vulnerability Database, CVE-ID CVE-2023-4863. 2023. URL: `https://nvd.nist.gov/vuln/detail/CVE-2023-4863`.

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

[4] D. Lehmann, J. Kinder, and M. Pradel. "Everything old is new again: Binary security of WebAssembly." In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 217–234.

[5] *Memory Safety*. Accessed on March 14, 2024. URL: `https://www.chromium.org/Home/chromium-security/memory-safety/` (visited on 03/14/2024).

[6] Qualcomm Technologies, Inc. *Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions*. White Paper. Accessed: 2023-12-14. 2017. URL: `https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf`.

[7] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. "AddressSanitizer: A fast address sanity checker." In: *2012 USENIX annual technical conference (USENIX ATC 12)*. 2012, pp. 309–318.

[8] K. Serebryany, C. Kennelly, M. Phillips, M. Denton, M. Elver, A. Potapenko, M. Morehouse, V. Tsyrklevich, C. Holler, J. Lettner, et al. "GWP-ASan: Sampling-Based Detection of Memory-Safety Bugs in Production." In: *arXiv preprint arXiv:2311.09394* (2023).

[9] L. Szekeres, M. Payer, T. Wei, and D. Song. "Sok: Eternal war in memory." In: *2013 IEEE Symposium on Security and Privacy*. IEEE. 2013, pp. 48–62.

[10] G. Thomas. *A proactive approach to more secure code*. Accessed on March 14, 2024. URL: `https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/` (visited on 07/16/2019).

[11] J. Vander Stoep and C. Zhang. *Queue the Hardening Enhancements*. Accessed on March 14, 2024. URL: `https://security.googleblog.com/2019/05/queue-hardening-enhancements.html` (visited on 05/09/2019).