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

First and foremost, I would like to thank my supervisor, Pramod Bhatotia. My academic life would have been quite different without his guidance and support. Since writing my Bachelor's thesis in 2021, his chair has been a welcoming place, allowing me to grow personally and as a researcher while opening many doors for the future. I am also very grateful to my advisor, Dimitrios Stavrakakis, for answering my questions about memory safety, giving me feedback on my work, and helping me debug weird measurements. Thanks to Dennis Sprokholt for his feedback on my formalizations.

To my friends, who have been with me throughout my master's thesis and beyond in both academic and personal journeys: Thank you for making this time enjoyable. While I cannot name everyone here, I want to extend special thanks to Andreas and Christoph for their feedback on this thesis.

During my last semester break, I had the opportunity to do an internship at Huawei's Helsinki System Security Lab. I gained extensive knowledge in compilers, WebAssembly, and security, greatly enhanced by the collaboration with my colleagues Antti, Janne, Carlos, Rémi, Valentin, and my manager, Jan-Erik, despite our short time sharing the office.

Lastly, I'd like to thank my family. Their support enabled me to study in Munich, and I am forever thankful for the unconditional support they have provided me.

Abstract

In this thesis, we investigate the design and implementation of an extension to WebAssembly (WASM) aiming to prevent the issue of memory safety vulnerabilities, particularly in languages like C and C++ that compile to WASM. Despite WASM's sandboxing feature that isolates applications from other instances and the host, these languages are still prone to memory safety bugs due to their lack of memory safety provided by the type system or prevalent libraries. This thesis introduces a minimally invasive extension to WASM designed to allow implementations to utilize diverse hardware- or software-based memory safety mechanisms.

Our work includes a complete compiler toolchain for C/C++ in LLVM, hardening programs, and providing spatial and temporal memory safety for heap and stack allocations. We showcase an implementation utilizing ARM's hardware-based Memory Tagging Extension (MTE), that offers a high-performance, low-overhead solution for spatial and temporal memory safety issues and is compatible with real-world performance requirements.

We further explore the possibility of integrating MTE into WASM's sandboxing mechanism, improving the performance of programs relying on expensive software-based bounds checks. The empirical evaluation on actual hardware platforms validates our proposed system's practicality and performance advantages.

Our work enhances WebAssembly with memory safety guarantees by introducing a generic, minimally invasive extension with low overhead. It sets a groundwork for further studies, suggesting directions for improving compatibility, optimizing performance, and incorporating various memory safety mechanisms.

Zusammenfassung

In dieser Arbeit untersuchen wir den Entwurf und die Implementierung einer Erweiterung von WebAssembly (WASM), die darauf abzielt, das Problem der Speichersicherheitsschwachstellen zu verhindern, insbesondere in Sprachen wie C und C++, die nach WASM kompiliert werden. Trotz der Sandboxing-Funktion von WASM, die Anwendungen von anderen Instanzen und dem Host isoliert, sind diese Sprachen immer noch anfällig für Speichersicherheitsfehler, da sie keine Speichersicherheit durch das Typsystem oder gängige Bibliotheken bieten. In dieser Arbeit wird eine minimal invasive Erweiterung von WASM vorgestellt, die es Implementierungen ermöglicht, verschiedene hardware- oder softwarebasierte Speichersicherheitsmechanismen zu nutzen.

Unsere Arbeit umfasst eine vollständige Compiler-Toolchain für C/C++ in LLVM, die Härtung von Programmen und die Bereitstellung von räumlicher und zeitlicher Speichersicherheit für Heap- und Stack-Allokationen. Wir stellen eine Implementierung vor, die ARMs hardwarebasiertes Memory Tagging Extension (MTE) verwendet, das eine leistungsstarke Lösung mit geringem Aufwand für räumliche und zeitliche Speichersicherheit bietet und mit realen Leistungsanforderungen kompatibel ist.

Außerdem untersuchen wir die Möglichkeit, MTE in den Sandboxing-Mechanismus von WASM zu integrieren, um die Leistung von Programmen zu verbessern, die auf teure softwarebasierte Bound Checks angewiesen sind. Die empirische Evaluierung auf aktuellen Hardware-Plattformen bestätigt die Praktikabilität und die Leistungsvorteile des von uns vorgeschlagenen Systems.

Unsere Arbeit erweitert WebAssembly um Garantien für Speichersicherheit, indem wir eine generische, minimal invasive Erweiterung mit geringem Overhead einführen. Sie bildet die Grundlage für weitere Studien und zeigt Wege zur Verbesserung der Kompatibilität, zur Optimierung der Leistung und zur Integration verschiedener Speichersicherheitsmechanismen auf.

Contents

1. Introduction

In recent years, WASM [10] has gained prominence [17] as a versatile compilation target, serving not only to web-based applications but also to a broader spectrum of use cases [37]. At its core, WASM is engineered as an efficient compilation target for high-level, compiled languages such as C and C++. A fundamental aspect of its design is its linear memory model, which allows these languages to efficiently compile to WASM and WASM to efficiently compile to various architectures.

While WebAssembly provides a sandbox for untrusted 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.

Recent ISA extensions, such as ARM's Pointer Authentication (PAC) [25] and Memory Tagging Extension (MTE) [4], offer promising, high-performance solutions by providing building blocks that incur low performance overhead. These hardware extensions are designed to effectively address memory safety concerns.

We introduce an extension to WASM to address spatial and temporal memory safety bugs with a prototype implementation utilizing MTE to efficiently implement the safety guarantees. Our approach requires no modification to the source code, as we provide a complete compiler toolchain and standard library that can be used to harden unmodified C/C++ programs. The core contributions of this thesis are outlined as follows:

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

Compiler Toolchain: A compiler toolchain that transparently transforms unmodified programs to provide spatial and temporal memory safety for stack and heap allocations.

Runtime: A WASM runtime with support for our WebAssembly extension and modified WASM compiler that utilizes ARM's MTE.

Bounds Checks with MTE: An implementation to eliminate expensive software-based bounds checks for 64-bit WASM programs.

Evaluation on real hardware: Evaluation of our implementation on ARM hardware, including performance and memory overheads as well as security guarantees, and MTE as implemented in actual production hardware.

2. Background

In this chapter we discuss the necessary background on which our work builds upon. We start by discussing WASM, memory safety in the context of WASM and ARM's PAC and MTE hardware extensions.

2.1. WebAssembly

WebAssembly [10], initially designed as an alternative, high-performance compilation target to JavaScript, continues to be applied in various use cases. WASM was carefully designed to allow 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 without being forced into an unnatural idiom. Languages may ship their allocators, garbage collectors, and layout data structures as efficiently as possible. This linear memory can then be mapped directly to the virtual memory on the host.

Structured control flow: In WASM, unstructured control flow is not allowed by design. WASM uses indices into type- and bounds-checked tables instead of raw function pointers to make indirect function calls. For jumps, WASM provides a set of well-defined control flow constructs. This reduces the attack surface of programs compiled to WASM and aids code generation.

Stack machine: Since compilation targets offer different sets of registers, WASM does not expose registers but operates on a typed stack instead. The stack can be verified to be well-typed and compiled to diverse targets, such as register machines or intermediate representations (IRs) in a single pass. No assumptions about the number of guest registers are made, as these can vary between different architectures and runtimes, as they might reserve some registers for their own use (e.g., 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. Different proposals add vector-, garbage-collected-, and

reference types covering other use cases. There is no distinction between pointer- and integer types. While this loses information present in the original program, such as pointer provenance, and prevents some optimizations, this is not considered a problem. In most cases, WASM is generated by an optimizing compiler, which has already performed optimizations relying on analyses such as alias analysis. The design of WASM allows for an efficient compilation to this format, which requires little optimization for the runtime compiling to native code.

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 Linux containers in Docker[1], or as an isolation mechanism to enable running untrusted code 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.

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

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 Memory Management Unit (MMU) will catch accesses into these pages, and the operating system will deliver a segmentation fault to the runtime, which will deliver a trap to the WASM program.

While the design of WebAssembly is designed to prevent malicious or erroneous programs from compromising the host, buggy programs are still vulnerable to classical memory safety errors discussed in section 2.2, 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 (1) out-of-bounds or (2) dangling pointers, which are the fundamental attack primitives enabling a whole class of attacks on a vulnerable or

[1] `https://docs.docker.com/desktop/wasm/`

buggy program [31]. Approaches to tackle these issues exist in several forms. Several studies have shown that in large software projects, memory safety bugs make up between 70 % and 75 % of all issues [15, 33, 34].

Lehmann et al. show that while some attack surfaces, such as those jumping to arbitrary addresses or injecting shellcode, are mitigated by the design of WebAssembly, buffer overflows or dangling pointer accesses are still possible [14]. Since WebAssembly does not provide separate read-only memory regions, this opens up other surfaces, allowing attackers to overwrite static data since compilers place them in the linear memory with both read and write permissions. Crucially, neither fundamental attack primitives (1) nor (2) are prevented by WebAssembly and can form the basis of an exploit.

Programs may be written in managed languages that prevent these attacks by not providing raw pointer accesses, such as Java, Python, JavaScript, or others. In these languages, memory access is performed through bounds-checked arrays or managed objects, such as classes. A garbage collector is responsible for cleaning up dangling objects. 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 invalid programs, e.g., programs containing dangling references or raw pointer accesses. An escape hatch in the form of `unsafe` exists, which allows dropping down to the level of C and directly manipulating 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.

2.2.1. Software-Based Mitigations

To detect and mitigate memory safety bugs in languages that do not provide safety guarantees at the language level, numerous approaches have been proposed [28, 29, 20, 30]. Checks can be inserted automatically at the compiler level, an approach chosen by Address Sanitizer (ASan) and Hardware-Assisted Address Sanitizer (HWASan) [28, 30]. ASan incurs significant overhead, on average 73%, which is too high 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 does not provide complete protection for a single process [29]. On a large scale, however, this approach allows for discovering real-world bugs that may not be triggered by testing or fuzzing workloads.

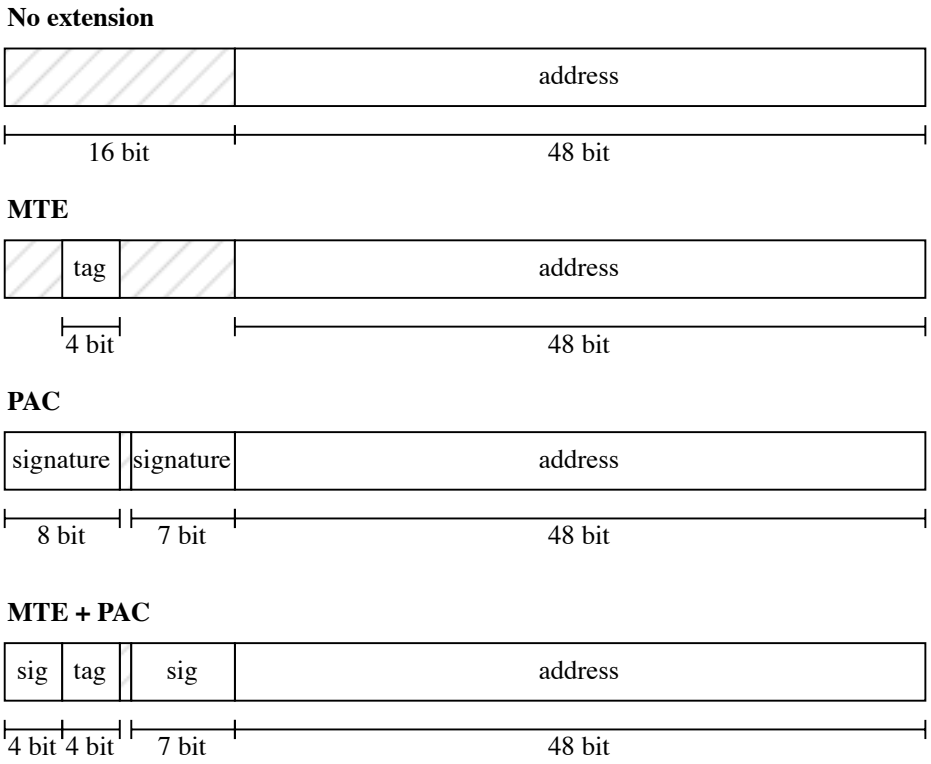


Figure 2.1.: Pointer layout on aarch64 in Linux with and without MTE and PAC enabled.

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 developed several hardware extensions designed as an efficient foundation for memory safety. They provide security primitives that compilers or programmers can use to provide full or partial memory safety to programs while having a small enough additional memory footprint and performance overhead to ship these solutions in production.

In figure 2.1, we show the layout of pointer bits used for address translation in aarch64, the 64-bit variant of the ArmV8 instruction set [3], when running on Linux. 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 are unused for further address translation. Hardware extensions such as Top Byte Ignore (TBI), MTE (section 2.3.1), or PAC (section 2.3.2) utilize those unused bits to store metadata.

2.3.1. Memory Tagging Extension (MTE)

ARMs MTE, available from ArmV8.5, provides a building block to prevent spatial and temporal memory safety violations [4]. MTE implements a lock-and-key mechanism where memory regions can be tagged with one of 16 distinct tags, and memory access is only allowed using pointers with the corresponding keys.

The locking mechanism is implemented by storing a 4-bit tag in bits 56–59 of an address (referred to as the logical tag). Accordingly, a tag is assigned to memory with a granularity of 16 bytes (referred to as the allocation tag).

On Linux, each process can configure MTE by switching between the following modes:

- **Disabled:** MTE is disabled, and no tag checks are performed.
- **Synchronous:** Tag mismatches cause a hardware fault on instruction retirement, and a segmentation fault is delivered to the application. The faulting instruction cannot read the affected memory location, or the update is not observable in the case of writes.
- **Asynchronous:** Tag mismatches do not cause an immediate hardware fault. Instructions may be able to read the memory location regardless of tag mismatches, or the update may be observable in the case of writes. The fault is delivered after the instruction has retired in the form of a CPU flag. The kernel will check this flag at the next context switch and deliver a segmentation fault to the application.

Temporal and Spatial Memory Safety

MTE can provide spatial memory safety by assigning different tags to adjacent regions and temporal memory safety by retagging freed memory. An example can be seen in figure 2.2.

2.3.2. Pointer Authentication (PAC)

PAC [25] introduces primitives to prevent attackers from modifying pointers stored in memory. The extension provides three instructions for signing pointers, authenticating, or stripping the signature from pointers. PAC places the signature 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. The signature can be between 7 and 16 bits long.

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

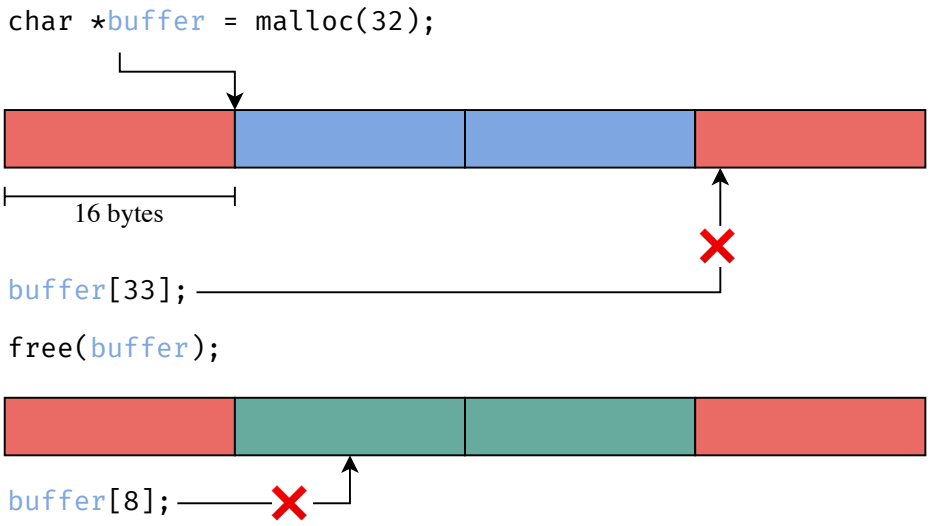


Figure 2.2.: Example of a heap allocation protected by MTE. The pointer returned by malloc and the allocation it points to are tagged with the same tag (■), while the surrounding memory is tagged with a different tag (■). The hardware checks for tag mismatches and thus prevents an out-of-bounds error (`logical_tag(buffer)` = ■ ≠ ■ = `allocation_tag(buffer[33])`). When freeing memory, the memory region is tagged with a new tag (■). This prevents use-after-free errors (`logical_tag(buffer)` = ■ ≠ ■ = `allocation_tag(buffer[8])`)

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. The extension provides `strip` instructions to remove the signature regardless of whether it is valid or not.

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 (see figure 2.1). The remaining bit 55 differentiates between lower and upper (kernel- and userspace) addresses.

3. Motivation

While WebAssembly provides strong safety and security guarantees, as discussed in section 2.1, they mainly guarantee safety for the host, but not for the program itself. In [14], Lehmann et al. show that while some attack surfaces, such as those 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. Such cases need to be mitigated at the language level by rewriting software in a safe language such as Rust, manually inserting bounds checks, which is error-prone, or inserting checks using the compiler and sanitizing the code.

Additionally, bugs like CVE-2023-4863 [7] continue to be exploited, showing that memory safety is not a solved problem. While they do not escape the WebAssembly sandbox, they pose a security risk to the programs themselves. 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 listing 1, the lack of bounds checks allows an attacker controlling the variable `input` to write beyond the allocation of `buf` and overwrite `str`.

```
1 void foo(char *input) {
2     char buf[32];
3     const char str = "Hello, World!";
4     strcpy(buf, input);
5 }
```

Listing 1: Vulnerable overflow.

WASM engines use various techniques to protect their sandboxes against malicious code (see section 2.1.1). While virtual memory and guard pages are preferred for performance, some situations (e.g., running 64-bit WASM) require software-based bounds checks. This approach provides necessary security but comes at a performance cost. In our measurements, switching to 64-bit WASM resulted in a roughly 6–8 % overhead on out-of-order CPUs, which can speculate bounds checks, and 47 % overhead on in-order CPUs (see detailed evaluation in section 7.2). The fallback to software-based bounds checks is thus especially painful when running on low-power in-order cores

using 64-bit WASM or in environments without an operating system, such as embedded devices. The results on the out-of-order CPUs are similar to a previous evaluation done by Szewczyk et al. [32].

4. Overview

In this thesis, we present a design for an extension that adds memory safety in WASM built on top of the WASM 64-bit memory proposal[1]. The extension is created to be minimally invasive and implementable using various techniques, including hardware extensions such as MTE or PAC, capability-based architectures like Capability Hardware Enhanced RISC Instructions (CHERI) [38], or software-based solutions similar to ASan [28] or HWASan [30] (see chapter 5).

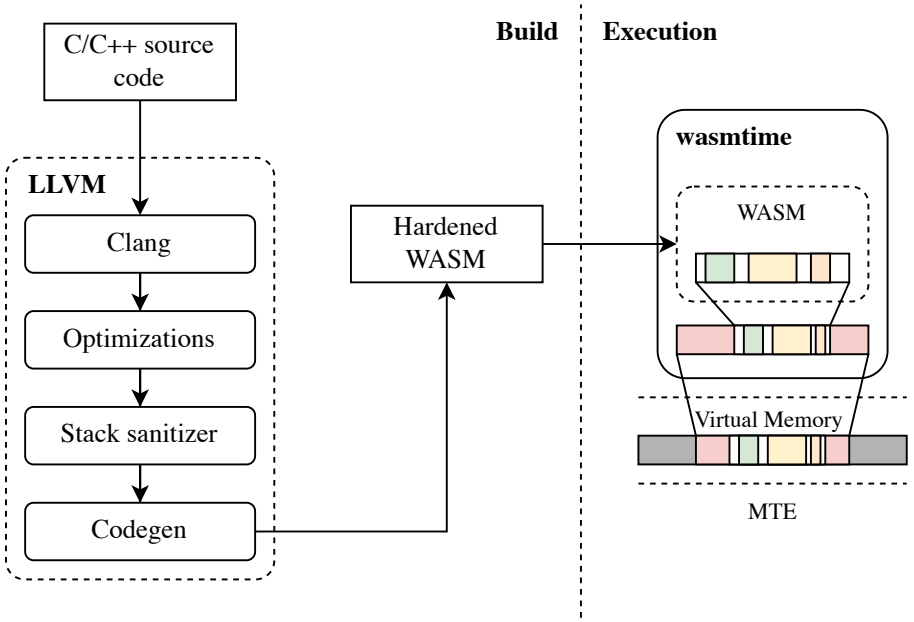


Figure 4.1.: Overview of the prototype implemented in this thesis.

In figure 4.1, we present our prototype of this design. Unmodified C/C++ source code is compiled using LLVM [13], where we implement a sanitizer that identifies and hardens stack allocations, along with a modified standard library based on WebAssembly System Interface (WASI) that protects heap allocations. LLVM then generates hardened

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

WASM binaries that can be run in wasmtime[2]. We modify wasmtime to process our WASM extension and implement it using MTE.

Additionally, we explore and implement a technique to efficiently sandbox WASM programs using MTE, eliminating expensive software-based bounds checks required for 64-bit WASM programs. This technique can be combined with the MTE-based memory safety implementation.

We analyze and benchmark various aspects of our implementation, including 32-bit and 64-bit WASM, and the MTE implementation on real hardware (see chapter 7).

[2] `https://wasmtime.dev/`

5. Design

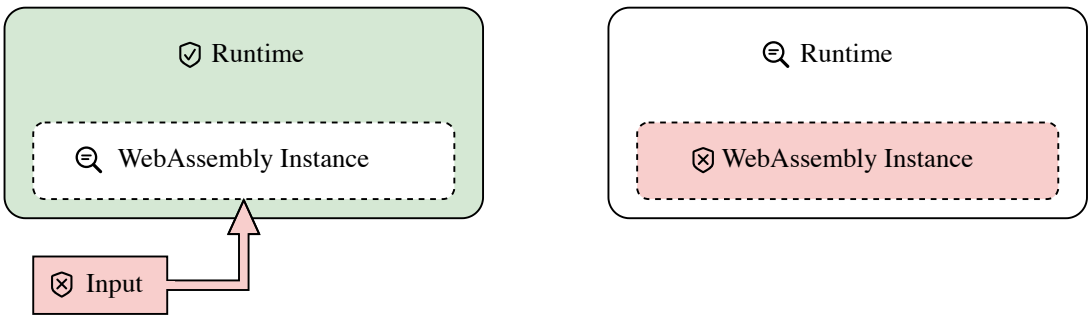
This chapter defines our threat model, WASM extension, and design to provide memory safety for programs compiled to WASM.

5.1. Threat Model

In our threat model (figure 5.1), we differentiate between two aspects of memory safety. In both models, we depict trusted components (marked with a) in green and untrusted components (marked with a) in red. We additionally highlight the component we are trying to harden (marked with a).

Internal Memory Safety: Ensures memory safety within the boundaries of a sandbox. Our point of view is from within a WebAssembly instance, we trust the runtime (and the host we are running on), but we do not trust external input (figure 5.1a).

External Memory Safety: Maintains the memory safety of the sandbox itself against potentially malicious programs. Our point of view is from the runtime; we trust the platform we are running on, but not the WebAssembly programs we are executing (figure 5.1b).



(a) Internal memory safety.

(b) External memory safety.

Figure 5.1.: Threat model for internal and external memory safety.

5.1.1. Internal Memory Safety

For internal memory safety, the program within the sandbox and its runtime, including its compiler, are considered trusted and assumed to be bug-free. Untrusted input (e.g., network data, file reads) originates from outside the sandbox and may be controlled by an attacker. This model mirrors the threat environment of a standard non-WASM program. Potential threats include:

- **Buffer overflows:** Attempts to access memory beyond allocated buffer boundaries.
- **Use-after-free:** Attempts to access deallocated memory.

As discussed in section 2.1, WebAssembly's design inherently mitigates some 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.

5.1.2. External Memory Safety

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

- **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 operating system and underlying target architecture are free of bugs that malicious targets might exploit. This does not include assumptions about potential spectre-like [12] attacks. The compiler needs to ensure that bounds checks are guarded against side-channel attacks.

Additionally, we do not consider exploits of the program running in the sandbox as vulnerabilities as long as they stay contained in the sandbox.

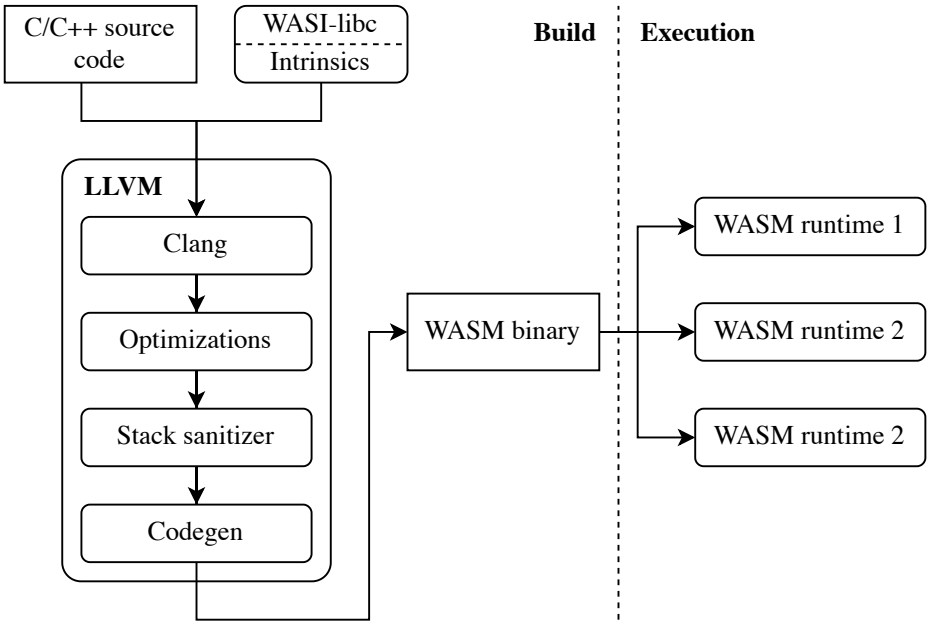


Figure 5.2.: Overview of the compilation and execution workflow.

5.2. Overview

Figure 5.2 presents an overview of our prototype. At build time, the unmodified C/C++ sources and a modified version of libc are compiled using LLVM [13]. After optimizations, a stack sanitizer analyzes all functions and inserts instrumentation as necessary. LLVM's backend then generates WebAssembly binaries that can be deployed and executed on various devices.

5.3. WebAssembly Extension

We designed an extension to WebAssembly that provides primitives to the modified standard library and the stack sanitizer to guarantee memory safety for selected allocations. Our extension builds on wasm64, the 64-bit variant of WebAssembly. We choose wasm64, because it uses a 64-bit integer index type, with 48 of those bits used to index memory. This allows us to allocate and store up to 16 bits of metadata per pointer.

For our extension, we propose the notion of abstract segments and tagged pointers. We introduce three new instructions that allow the creation of abstract segments and

tagged pointers from raw pointers. These pointers carry provenance and can only access the segment they were created with. Conversely, segments can only be accessed by the tagged pointer created with them rather than with raw indices without provenance. We introduce the following new instructions. Each instruction takes a constant unsigned offset o, which allows compilers to fold in constant offsets when manipulating segments.

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

In the following paragraph, we describe the new instructions in detail.

`segment.new`: Create a new, zeroed memory segment. This instruction takes two arguments: a memory index and a size. The instruction generates a new tag, assigns it to the piece of memory, and returns a tagged pointer that can be used to access the segment.

`segment.set_tag`: This instruction takes a memory index, a length, and a tagged pointer and applies the tag from the tagged pointer to the memory segment located at the index with the passed length. This can be used to move ownership from one segment to another or to merge segments.

`segment.free`: This instruction invalidates a segment by tagging the segment with a new, implementation-defined tag. This instruction takes two arguments: a memory index and a length. After this instruction, the tagged pointer being used to access the segment is no longer valid, and accessing the segment through it will result in a trap.

We also modify the semantics of existing load and store instructions. They still take an integer as an index, but we introduce provenance to integers, which we can track at runtime using the unused 16 upper bits in pointers. If a segment is accessed, the runtime will check that the tagged pointer is allowed to access the segment, i.e., the metadata matches the metadata created by the `segment.new` instruction. If this is not the case, the runtime throws a trap.

At startup, the linear memory consists of a single segment that can be accessed using untagged indices, allowing unmodified code to run under our new semantics without modifications. This design choice also 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 hardened malloc implementation, which prevents spatial and temporal safety bugs for heap-allocated memory. Additionally, we can analyze stack allocations to only harden those accessed using untrusted indices or escape our analysis, e.g., by taking their address and passing it to another function.

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

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

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

Figure 5.3.: Typing rules of the new instructions. For the definition of context C, see the WASM paper [10].

Alignment All segments are aligned to 16 bytes, corresponding to the alignment of MTE (see section 2.3.1). This is an implementation choice that may be changed once we support additional implementations. More details can be found in section 9.1.

5.3.1. Typing Rules

In figure 5.3, we extend the typing rules of the WASM paper [10] in the notation of Pierce [22]. 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 instruction can only be used when a memory is declared. The type $tf = t_1^* \rightarrow t_2^*$ describes how the instruction manipulates the operand stack. The instruction e expects an operand stack where it pops off t_1^* and pushes t_2^*.

$$\begin{aligned}\text{(store) } s &::= \{\ldots, \text{tag } \mathit{taginst}^*\}\\ \mathit{taginst} &::= b^*\end{aligned}$$

$$s; (\textbf{i64.const } k); (t.\textbf{load } a\ o) \hookrightarrow_i \textbf{trap} \quad \text{if } s_{\text{tag}}(i, k+o, |t|) \neq \text{tag}(k) \tag{5.1}$$

$$s; (\textbf{i64.const } k); (t.\textbf{const } c); (t.\textbf{store } a\ o) \hookrightarrow_i \textbf{trap} \quad \text{if } s_{\text{tag}}(i, k+o, |t|) \neq \text{tag}(k) \tag{5.2}$$

$$s; (\textbf{i64.const } k); (\textbf{i64.const } l); (\textbf{segment.new } o) \hookrightarrow_i s'; (\textbf{i64.const } t) \quad \text{if } t = \text{new\_tag}(k+o) \wedge s' = s \text{ with } s_{\text{tag}}(i, k+o, l) = t \tag{5.3}$$

$$s; (\textbf{i64.const } k); (\textbf{i64.const } t); (\textbf{i64.const } l); (\textbf{segment.set\_tag } o) \hookrightarrow_i s' \quad \text{if } s' = s \text{ with } s_{\text{tag}}(i, k+o, l) = t \tag{5.4}$$

$$s; (\textbf{i64.const } k); (\textbf{i64.const } l); (\textbf{segment.free } o) \hookrightarrow_i s' \quad \text{if } t = \text{free\_tag}(k+o) \wedge s' = s \text{ with } s_{\text{tag}}(i, k+o, l) = t \tag{5.5}$$

$$s; (\textbf{i64.const } k); (\textbf{i64.const } l); (\textbf{segment.new } o) \hookrightarrow_i \textbf{trap} \quad \text{otherwise} \tag{5.6}$$

$$s; (\textbf{i64.const } k); (\textbf{i64.const } t); (\textbf{i64.const } l); (\textbf{segment.set\_tag } o) \hookrightarrow_i \textbf{trap} \quad \text{otherwise} \tag{5.7}$$

$$s; (\textbf{i64.const } k); (\textbf{i64.const } l); (\textbf{segment.free } o) \hookrightarrow_i \textbf{trap} \quad \text{otherwise} \tag{5.8}$$

Figure 5.4.: Small-step reduction rules of the new instructions and added rules for load/stores. See the WASM paper [10] for the definitions of all rules and auxiliary constructs.

5.3.2. Small-Step Reduction Rules

In figure 5.4, we extend the small-step reduction rules from the WASM paper [10] using the notation established by Plotkin [23]. The lower portion of figure 5.4 presents new tag-aware load/store rules that take precedence over the existing ones and new rules for the introduced instructions.

To signal a trap, we reuse operators from the original WASM rules, including the

trap operator. The state, s, is augmented with a storage mechanism that assigns a tag t, to each 16-byte granule of memory. 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, i.e. *addr* mod $16 = 0$
- $t = \text{tag}(pointer)$: 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.

Figure 5.4 highlights the added and modified components in the rules. The added load-/store rules, specified in equations (5.1) and (5.2), enforce trapping on tag mismatches. Similarly, when an access spans memory regions with different tags, these rules trap. The rules for executing the new instructions, which modify the state by setting tags, are presented in equations (5.3) to (5.8). Equations (5.3) to (5.5) represent the default case where tags are assigned to aligned, in-bounds addresses, with equations (5.6) to (5.8) representing the cases with unaligned or out-of-bounds accesses. Each reduction rule is depicted with the operand stack's top and state s on the left-hand side, representing the pre-execution state, and the resulting stack and state after the execution of the instruction on the right-hand side. For the rules with $\hookrightarrow_i$, the i represents the instance the instruction is executed in.

5.3.3. Example

We will demonstrate our WASM extension using the C snippet in listing 2, which allocates 64 bytes on the stack.
This requires the compiler to instrument the stack allocation, create a new segment, and free the segment before returning to the caller, i.e., giving ownership of the stack slot back to the stack frame, as demonstrated in listing 3.

The compiler allocates the slot for `buf` on the stack, decrementing the global `$__stack_pointer` acting as the stack pointer (lines 2 to 5). Then, a new segment of size 64 is created, and the tagged pointer to it is stored in the local `$buf` (lines 8 to 10).

```c
void foo() {
    char buf[64];
    // ...
    return;
}
```

Listing 2: Example of a C program allocating 64 bytes on the stack.

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

;; create a 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
```

Listing 3: Generated WASM for code from listing 2.

Before returning, the segment is retagged using the stack pointers tag, i.e., restoring the previous tag and allowing access through the stack pointer (lines 15 to 18). Then, the stack pointer is reset, freeing the stack frame (lines 21 to 24).

5.3.4. Heap Safety

The memory allocator needs to be aware of segments to provide heap safety. When allocating memory, it aligns the requested size to 16 bytes, creates a segment, and returns the corresponding tagged pointer. This prevents overflows from corrupting allocator metadata or other memory segments. A modified allocator implementation looks conceptually similar to the snippet in listing 4.

```
1 void *malloc(size_t length) {
2     void *chunk = /* perform allocation */;
3     return __builtin_wasm_segment_new(chunk, length);
4 }
```

Listing 4: Example of a malloc implementation utilizing the memory safety extension.

When compiled to WASM, we see just three new instructions added to the generated code in listing 5 (lines 6 to 8). This proves that our extension is minimally invasive, as the calling code does not need to be changed and will continue to work as-is. Similarly, when freeing or reallocating memory, the allocator needs to ensure that the no longer valid memory is retagged.

```
1 (func $malloc (param $length i64) (result i64) (local $chunk)
2     ;; perform allocation and place the result in $chunk
3     ;; ...
4
5     ;; create a segment
6     local.get $chunk
7     local.get $length
8     segment.new
9     ;; implicit return
10 )
```

Listing 5: Generated WASM for code from listing 4.

```
allocsToInstrument ← ∅
for alloc ∈ allocations do
    if escapes(alloc) then
        allocsToInstrument ← allocsToInstrument ∪ { alloc }
    else if isUsedByUnsafeGEP(alloc) then
        allocsToInstrument ← allocsToInstrument ∪ { alloc }
    end if
end for
for alloc ∈ allocsToInstrument do
    insertTaggingCode(alloc)
    insertUntaggingCode(alloc)
end for
```

Figure 5.5.: Algorithm to detect and harden safe and unsafe stack allocations.

5.3.5. Stack Safety

For stack safety, we create segments from stack slots when entering a function. Before returning, all stack slots are untagged and reassigned to the stack frame. This allows other functions to use the memory and prevents stack slots from being accessed after returning from a function.

However, only some stack allocations need to be turned into segments. We can omit allocations that do not escape or are only accessed using statically verifiable indices. Creating segments for these would result in excessive runtime and memory overhead, as each allocation would need to be aligned to 16 bytes and processed when entering and returning from a function.

To address this, we design an algorithm (figure 5.5) that identifies safe memory regions within the stack which do not require protection, thus avoiding creating segments for the slots mentioned above. In figure 5.5, we present a simplified version of our algorithm. We iterate over all stack allocations and check if the allocation (a) escapes the function or (b) is indexed into using an unsafe `getelementptr` (GEP) instruction.

5.3.6. Example

We demonstrate the algorithm using the functions in listing 6. In the code example, `i` (line 2) is safe, as its address is not used in a potentially unsafe address computation and does not escape. The variables `bytes_read` and `buf` (lines 3 and 4) are deemed unsafe as their address escapes (line 5). Additionally, `buf` is accessed using an untrusted index (line 6).
In function `bar`, `buf` also needs to be instrumented as it escapes: the pointer to it

```
1 char foo(int index) {
2   int i = 0; // safe
3   int bytes_read = 0; // unsafe
4   char buf[32]; // unsafe
5   read_input(buf, &bytes_read);
6   return buf[index];
7 }
8
9 char *bar() {
10     char buf[32]; // unsafe
11     return buf;
12 }
```

Listing 6: Example code for safe and unsafe stack slots.

is returned to the caller (line 11). Any attempt to dereference the value returned by `bar` is undefined behavior and will be caught by our instrumentation, preventing difficult-to-debug bugs or potential vulnerabilities.
This algorithm effectively balances the need for stack safety with performance and memory efficiency constraints.

6. Implementation

We implement our prototype in the LLVM framework, wasi-libc, and the wasmtime WebAssembly runtime. The following sections detail the specific modifications and extensions to each component and some implementation choices and details to tackle specific problems.

6.1. LLVM

We choose LLVM as our compiler, from C/C++ to WebAssembly. We modify the existing WASM backend and add support for the extension described in section 5.3, allowing LLVM to emit the new instructions.

6.1.1. LLVM IR

In the middle end, we introduce three new intrinsic functions that correspond and are lowered to our WASM instructions by the backend.

`ptr @llvm.wasm.segment.new(ptr, i64)`
Takes an untagged pointer and a size, creates a new segment and returns the tagged pointer to it.

`void @llvm.wasm.segment.set_tag(ptr, ptr, i64)`
Takes a pointer to a memory region we want to retag, a pointer containing the new tag, and a size, and sets the tag for the memory segment.

`void @llvm.wasm.segment.free(ptr, i64)`
Takes a tagged pointer and a size and assigns a new tag to the region, ensuring the tagged pointer cannot be used to access it anymore.

The clang front end or a sanitizer pass can insert calls to these intrinsic functions. Listing 7 shows how we lower a function that allocates 32 bytes on the stack to LLVM IR.

```
define hidden signext void @foo(i32 %index) {
entry:
  %arr = alloca [32 x i8], align 16
  ; create a new segment
  %1 = call ptr @llvm.wasm.segment.new(ptr %arr, i64 32)

  ; do some work

  ; return ownership of segment to stack
  call void @llvm.wasm.segment.set.tag(ptr %1, ptr %arr, i64 32)
  ret void
}
```

Listing 7: Code generated for a function that allocates 32 bytes on the stack.

Table 6.1.: Flags added to LLVM.

Flag	Description	Required Flags
`-mmem-safety`	Enable the memory safety extension.	
`-fsanitize=wasm-memsafety`	Enable the stack sanitizer.	`-mmem-safety`

6.1.2. LLVM Sanitizer Pass

In LLVM, we introduce a WASM-specific sanitizer pass that can be enabled via a compiler flag (see table 6.1), designed to provide memory safety for stack allocations when compiling to WebAssembly. This sanitizer analyzes functions for stack allocations, applies padding, and inserts calls to the intrinsics described in section 6.1.1 to create segments, as discussed in section 5.3.5. The pass runs after all optimizations, ensuring we do not block passes that might remove stack allocations, such as `mem2reg`.

For each hardened allocation, we insert a call to the `segment.new` intrinsic after the allocation and replace all uses of the pointer to the allocation with our tagged pointer. Before returns or tail calls, we need to return ownership of the segment to the stack frame, i.e. retagging using the stack pointers tag. This serves two purposes: (1) subsequent code is able to access the memory through the stack pointer without a tagged pointer, and (2) use-after-return errors are caught, as the tagged pointer is not able to access the segment.

6.1.3. C extension

To create and manipulate segments manually, e.g., to build a segment-aware memory allocator, we expose the memory safety features to C in the form of built-in functions

Table 6.2.: Flags added to wasmtime.

Flag	Description	Required Flags
`-C mte`	Enable the use of MTE.	
`-C mte-bounds-checks`	Use MTE for bounds checks.	`-C mte`
`-W mem-safety`	Enable the memory safety extension.	`-C mte -W memory64`

that clang lowers to calls to the corresponding LLVM intrinsics.

```
void *__builtin_wasm_segment_new(void *, unsigned long);

void __builtin_wasm_segment_set_tag(void *, void *, unsigned long);

void __builtin_wasm_segment_free(void *, unsigned long);
```

The functions can be used as regular functions in C code, as illustrated in listing 8.

```
1 void *my_malloc(unsigned long size) {
2   void *memory = malloc(size);
3   return __builtin_wasm_segment_new(memory, size);
4 }
```

Listing 8: Example of how a built-in function can be called from C.

6.2. WASI Libc Modifications

To allow us to run applications relying on libc on wasm64, we port the WebAssembly System Interface (WASI) and wasi-libc to wasm64. This work includes changing size and pointer types from 32 to 64 bits.

We modify dlmalloc, the default allocator in wasi-libc, to provide memory safety for heap allocations. We insert calls to the built-in functions exposed to C as necessary, creating memory segments and returning tagged pointers to these segments. This protects both allocator metadata and adjacent allocations from being accessed or modified through heap overflows. When freeing or reallocating memory, segments are freed, ensuring temporal safety.

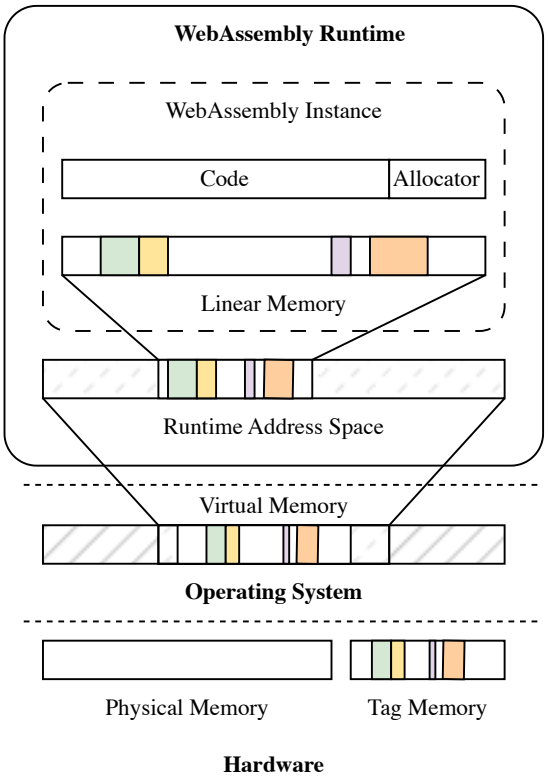


Figure 6.1.: The memory safety extension, as implemented in wasmtime using MTE. Memory segments are tagged, with different colors representing different tags. The processes virtual memory maps to physical memory and tag memory, which stores the tags assigned to memory granules.

6.3. Internal Memory Safety

We implement our prototype in wasmtime[1], a WebAssembly runtime mainly focusing on speed and correctness, written in Rust. Wasmtime features an optimizing compiler, cranelift[2], and its own IR, namely Cranelift IR (CLIF).

In figure 6.1, we present an overview of our implementation of the memory safety extension in wasmtime using MTE. We modify wasmtime and its supporting libraries and extend it with support to parse and process the memory safety extension described in section 5.3. We add support for MTE in the form of new instructions and lowering rules to cranelift, allowing wasmtime to generate MTE instructions when compiling

[1] `https://wasmtime.dev/`
[2] `https://cranelift.dev/`

Table 6.3.: Default tag t for the linear memory.

Memory Safety	MTE Sandboxing	Default Tag
No	No	$t \in \{0\}$
No	Yes	$t \in \{1, \ldots, 15\}$
Yes	No	$t \in \{0\}$
Yes	Yes	$t \in \{1\}$

for a target that supports them. We run in MTE synchronous mode to catch and stop memory safety violations before their effects become observable to the violating or any other process. These features can be enabled using the flags described in table 6.2.

The memory is tagged by default with one of the tags described in table 6.3, depending on the configuration. This tag is then stored in the heap base pointer of the WASM instance, allowing memory accesses through the pointer.

For MTE, we store the logical tag in the upper, unused bits of the WASM index. This index becomes a valid pointer once it is translated to an address by adding it to the heap base address.

6.3.1. Tagging Memory

When setting an allocation tag, we have four choices:

1. `stg`: Setting the tag for a single tag granule,
2. `st2g`: Setting the tag for two tag granules,
3. `stzg`: Setting the tag for a single granule and zeroing the granule.
4. `stgp`: Setting the tag for a single granule and storing a pair of registers.

To ensure new segments are always zeroed, options (1) and (2) require an additional memset. Our benchmarks for all four variants (see section 7.5) show `stzg` as the variant with the lowest overhead while also clobbering one less register compared to `stgp`.

6.3.2. Lowering WASM to machine code

In the following paragraphs, we describe how we lower each of our instructions to an MTE backend.

`segment.new` : To create a new segment, we (1) check that the requested segment is inside the linear memory of the guest, (2) generate a random logical tag and insert it into the index, and (3) set the allocation tag for the segment. This involves generating a loop that iterates over the size of the segment and setting the tag using `stzg`, which also zeroes the memory.

`segment.set_tag` : To change ownership of a segment, we (1) check that the requested segment is inside the linear memory of the guest and (2) set the new allocation tag for the segment. Here, we do not need to create a new, random tag, as we have passed a predefined tag.

`segment.free` : To invalidate a segment, we (1) check that the requested segment is inside the linear memory of the guest and (2) set the default allocation tag for the segment. The default tag depends on the configuration and can be taken from table 6.3.

We implement several optimizations to ensure our generated code runs efficiently. When setting the allocation tag for a segment, we generate a loop iterating over the size of the segment. If the size of the loop is known at compile time, we unroll the loop to tag up to 160 bytes per iteration to avoid branch instructions. We choose this tradeoff between code size and reducing the number of branch instructions executed.

6.3.3. Migration of the Linear Memory

When resizing linear memory, the runtime must relocate the existing contents and the MTE tags if active. We have two primary migration strategies (see figure 6.2).

Temporarily disable MTE: Disable MTE globally, copy memory and tags, then re-enable MTE. This compromises memory safety during the migration for other threads in the same process relying on MTE and can thus only be used if only a single WASM instance is running.

Copy data and tags: Load the memory tag for each 16 byte granule. Use the tag to safely load and store data between memory regions, then set the tag for the new granule.

We have measured the performance of both approaches in section 7.5.4. After taking the performance difference into consideration, we chose the second approach, as the approach to temporarily disable MTE relies on only a single thread in the process requiring MTE and is only marginally faster on some CPUs and slower on others.

```
disable_mte();
memcpy(from, to, len);
while from != end {
  asm!(
    "ldg {tag}, [{from}]",
    "stg {tag}, [{to}], #16",
    tag = out(reg),
    from = in(reg) from,
    to = inout(reg) to,
  );
  from = from.add(16);
}
enable_mte();
```

(a) Code generated for the first migration approach.

```
while from != end {
  asm!(
    "ldg {from}, [{from}]",
    "ldg {to}, [{from}]",
    "ldp {val1}, {val2}, [{from}]",
    "stgp {val1}, {val2}, [{to}], #16",
    from = in(reg) from,
    to = inout(reg) to,
    val1 = out(reg) _,
    val2 = out(reg) _,
  );
  from = from.add(16);
}
```

(b) Code generated for the second migration approach.

Figure 6.2.: Variants on how to migrate tagged memory.

6.4. External Memory Safety

In figure 6.3, we show our approach to utilize memory tagging to replace software-based bounds checks while preserving external memory safety. The runtime assigns a tag (see table 6.3) to each instance on module instantiation. This tag is stored in the heap base address. Memory access translation then involves adding the accessed index to the tagged heap base address. The memory outside the linear memory, i.e., memory belonging to the runtime, is always tagged with the zero tag, ensuring MTE catches accesses outside the sandbox with minimal modifications to the runtime.

However, we face a limitation in the number of sandboxes for this approach. Since MTE only offers up to 16 distinct tags, we are limited to up to 15 different sandboxes within one process, as we need to reserve one tag for the runtime.

We can combine this approach with the memory safety extension (see section 6.3) by further restricting the number of sandboxes in one process and thus freeing up tag bits for the memory safety extension. While it would be possible to assign up to three MTE tag bits for sandboxing, we allocate three bits for the memory safety extension, thus just allowing a single instance to run in the same process. We designate the lowest tag bit to determine whether memory belongs to the runtime or the linear memory. Since we reserved the zero tag for the runtime, we tag the linear memory with the tag 1. The remaining three tag bits used for the memory safety features can be generated by `segment.new`. This results in memory indices always having an even logical tag, with the actual memory locations being assigned an odd allocation tag.

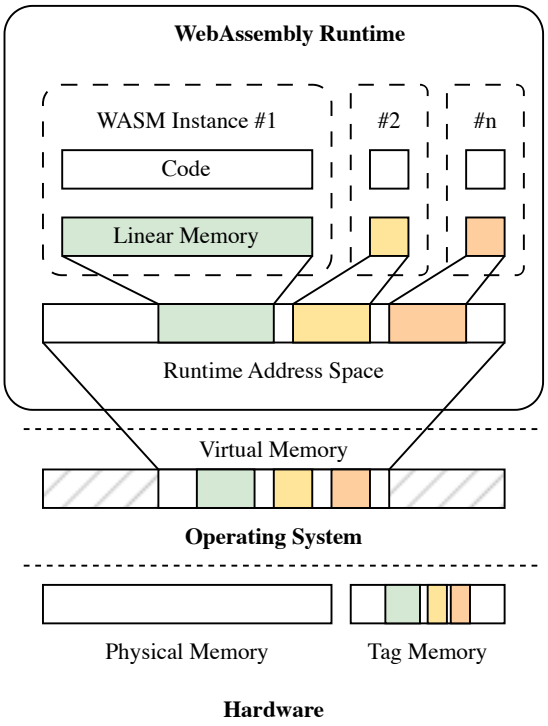


Figure 6.3.: Bounds checks as implemented in wasmtime using MTE. Each instances linear memory is assigned a unique tag, which re represent using different colors. The virtual memory of the process maps to physical memory and tag memory, which stores the tags assigned to memory granules.

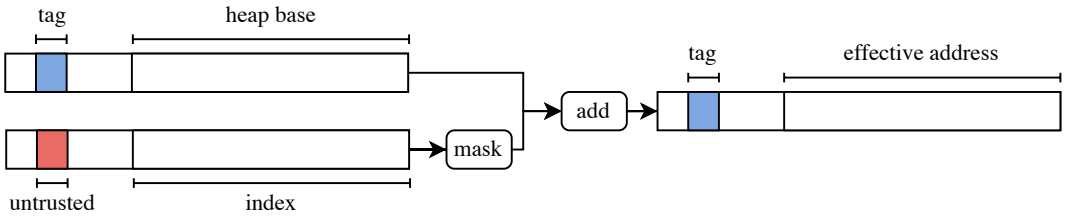
This approach comes with the following challenges:

1. Adding a tagged user pointer to the heap base address should be performant and result in the correct tag for the respective memory.
2. Untrusted WASM modules should not be able to forge tags that allow them to access memory beyond their allocated sandbox.

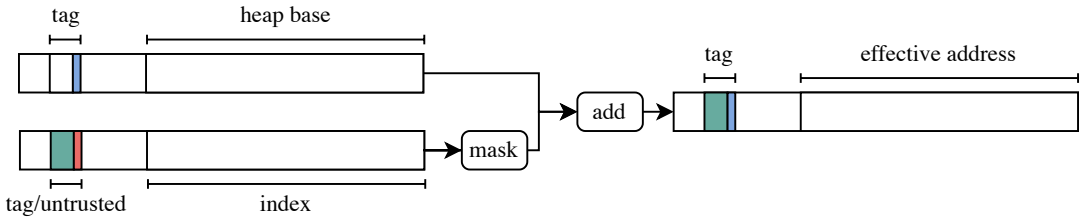
As accesses are performed through integers, which can be arbitrarily changed by code, adding an untrusted index to the heap base can be exploited to arbitrarily set the resulting logical tag, potentially allowing code to escape the sandbox. To prevent this issue, we mask the tag bits allocated for the runtime from the index before address computation, as shown in figure 6.4. Figure 6.4a shows the masked bits when only MTE bounds checking is enabled. In this case, all tag bits are assigned for bounds checks and are masked out of the tag bits in the index. In figure 6.4b, we demonstrate what

Table 6.4.: Tag mask for the index.

Memory Safety	MTE Sandboxing	Tag Mask
No	No	–
No	Yes	`and 0xF0FF_FFFF_FFFF_FFFF`
Yes	No	–
Yes	Yes	`and 0xFEFF_FFFF_FFFF_FFFF`



(a) MTE bounds checks.



(b) MTE bounds checks with memory safety extension.

Figure 6.4.: Effective address calculation with MTE bounds checks.

happens when both MTE bounds checking and memory safety are enabled. Here, only the lowest bit is assigned for bounds checks, while the remaining three are assigned to the memory safety extension. In table 6.4, we present the mask used for each configuration.

Excluding Tags from Random Tag Generation If we are running with MTE bounds checks enabled, we need to ensure that certain tags cannot be generated by instructions such as `irg` (**i**nsert **r**andom ta**g**) or `addg` (**add** ta**g**). Excluded tags can be either specified as an exclude mask via an immediate parameter or an include mask via a `prctl` call. We choose the latter option and set the include mask at startup time. In table 6.5, we list the include masks and included tags for the instructions mentioned above. If the memory safety extension is disabled, none of these instructions will be emitted, so we do not need to exclude tags from being generated.

Table 6.5.: Included tags for `irg`, `addg` instructions depending on configuration.

Memory Safety	**MTE Sandboxing**	**Tag Include Mask**	**Included Tags**
No	No	–	–
No	Yes	–	–
Yes	No	`0xffff`	$t \in \{1, 2, \ldots, 15\}$
Yes	Yes	`0x5555`	$t \in \{2, 4, \ldots, 14\}$

7. Evaluation

7.1. Experimental Setup

We conduct our benchmarks on a Google Pixel 8 equipped with a Google Tensor G3 chip, including 1 × Cortex-X3 (2.91 GHz), 4 × Cortex-A715 (2.37 GHz), and 4 × Cortex-A510 (1.7 GHz) cores, with MTE enabled. See table 7.1 for details on the benchmarked cores. As of the time of writing, this is the sole commercially available device featuring MTE. To mitigate thermal throttling, we attach a cooling fan to the device. Each performance test is run on each CPU type available on the Tensor G3 chip by pinning it to a single respective core. We run the benchmarks from the PolyBench/C 3.2 suite [24], which includes 30 kernels from domains such as linear algebra, data mining, stencils, and medley operations.

Table 7.1.: Comparison of the benchmarked cores.

Spec	**Cortex-X3**	**Cortex-A715**	**Cortex-A510**
Cores	1	4	4
Pipeline	out-of-order	out-of-order	in-order
Superscalar	Yes	Yes	Yes
Architecture	ArmV9.2	ArmV9.2	ArmV9.2
Maxmimum Frequency	2.91 GHz	2.37 GHz	1.7 GHz
L1 I-Cache/D-Cache	64 KiB	32/64 KiB	32/64 KiB
L2 Cache	512 KiB – 1024 KiB	128 KiB – 512 KiB	128 KiB – 512 KiB
L3 Cache	512 KiB – 16 MiB	512 KiB – 32 Mib	256 KiB – 16 Mib

Table 7.2.: Runtime benchmarking configurations.

Variant	**Pointer Width**	**Memory Safety**	**MTE Bounds Checks**
wasm32	32-bit	No	No
wasm64	64-bit	No	No
mem-safety	64-bit	Yes	No
mte-bounds	64-bit	No	Yes
combined	64-bit	Yes	Yes

7.2. Performance Overheads

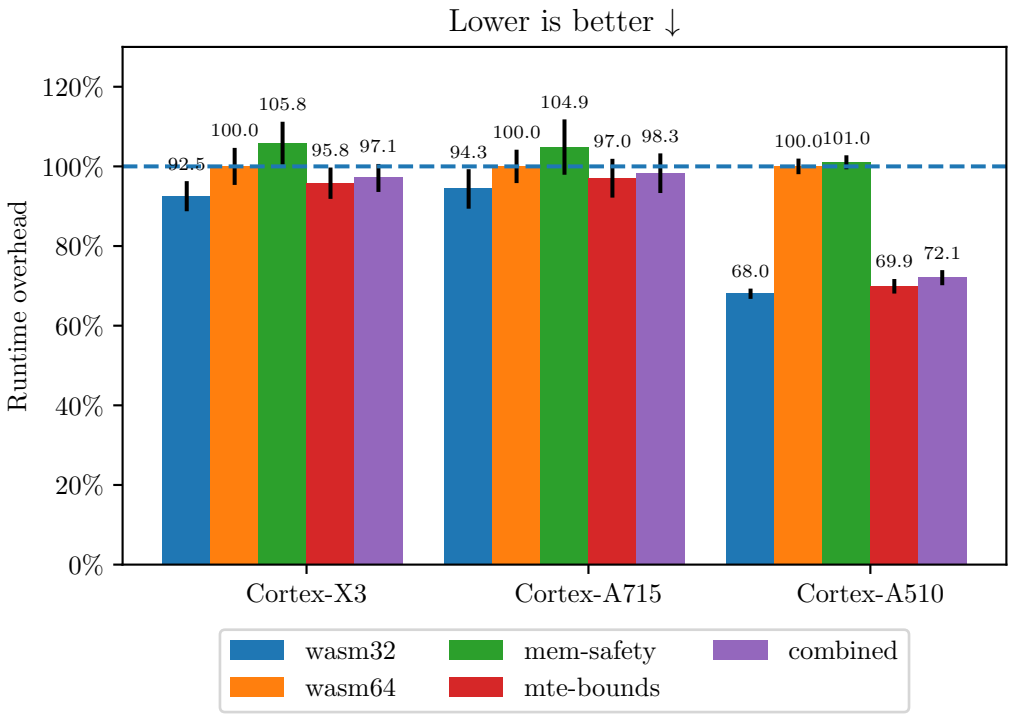


Figure 7.1.: PolyBench/C runtime overheads of different configurations described in table 7.2, normalized to wasm64.

Figure 7.1 illustrates the mean runtime overheads for PolyBench/C benchmarks for each CPU core available on the Tensor G3 chip. Compared to wasm64, our memory safety extension has a mean overhead of 5.8 % and 4.9 % on the two out-of-order high-performance cores. On the in-order Cortex-A510, we see an overhead of just 1.0 %. Generally, we observe that the overhead of bounds checks through the switch from wasm32 to wasm64 is lower on the out-of-order cores, as those can speculate on bounds checks, while the in-order cores cannot. This also explains the low overhead of our memory safety extension on the in-order cores, as we spend more time doing bounds checks than on the out-of-order cores.

When we replace software-based bounds checks with MTE sandboxing, we see the overhead largely disappearing. The remaining overhead can be explained through (1) the natural overhead that enabling MTE synchronous mode poses (see section 7.5.3)

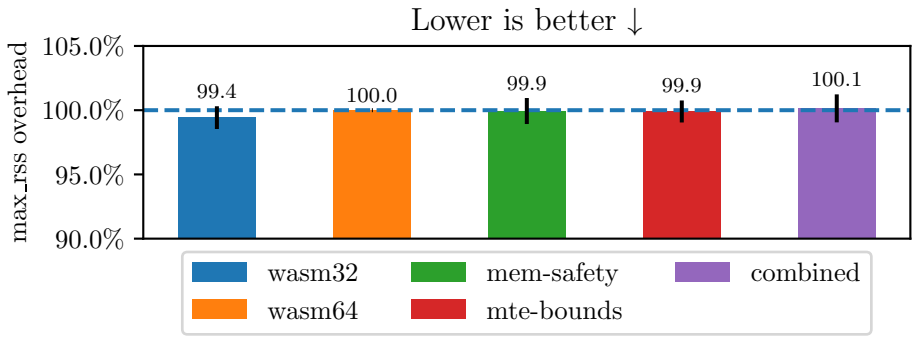


Figure 7.2.: PolyBench/C memory overheads of different configurations described in table 7.2, normalized to wasm64.

and (2) the fact that the linear memory needs to be tagged on program startup. This overhead is especially noticeable for short-running modules that require large amounts of linear memory.

Combining both MTE bounds and MTE memory safety, we see a slight increase from just MTE bounds, which is smaller than the jump from wasm64 to memory safety. The smaller increase is because we are not adding the MTE overhead, as MTE is already enabled. The jump we see is the result of additional instructions tagging the segments.

We could decrease the overhead even further by switching to MTE async mode, which is faster than sync mode (see section 7.5.3). However, this dramatically reduces the security guarantees provided by MTE, as illegal writes and reads may become observable. This disqualifies async MTE for bounds checking of WASM sandboxes, as attackers may carefully craft malicious code to escape their sandbox. For the memory safety extension, users may decide the additional risk is worth the reduced overhead, e.g., when the memory safety extension is not used as a primary defense mechanism but as a second layer or to find bugs in the wild.

7.3. Memory Overheads

Memory tagging incurs memory space overheads, particularly for small allocations due to the 16-byte alignment required for MTE. Our measurements did not show a significant difference in maximum resident set size (rss). This is because (1) safe allocations do not incur space overhead, and (2) for large allocations, the 16-byte alignment overhead is proportionally small. The main overhead in figure 7.2 is primarily due to the switch to wasm64, as pointer sizes are doubled.

7.4. Security Guarantees

We evaluate the security guarantees from the perspective of internal and external memory safety, as defined in section 5.1.

7.4.1. External Memory Safety

Running with MTE bounds checks, with MTE being configured to run in synchronous mode, prevents programs from escaping their sandbox. We prevent programs from forging tags by masking the respective tag bits before computing the effective address, as described in section 6.4. Other defense mechanisms, such as structured control flow, the typed stack, and function calls through typed and checked tables, remain unchanged by our implementation. 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.

Switching to async MTE mode is not feasible to retain external memory safety, as memory accesses outside the sandbox may become observable to the WASM module performing the illegal access and to other modules.

7.4.2. Internal Memory Safety

Our choice to utilize MTE results in low overhead, making it possible to deploy it in production. However, our approach does not provide complete memory safety for internal memory safety, as MTE provides a limited number of tags and should be used as a secondary, not primary, defense mechanism to harden applications in the wild. A tag collision means two memory regions could accidentally share the same tag, potentially leading to a missed security violation. For example, if a buffer overflow writes slightly beyond its intended bounds, but the adjacent memory has the same tag, MTE cannot detect the issue.

We can calculate the probability for a tag collision for $k = 2$ random tags according to figure 7.3, with $n = 16$ available tags for MTE bounds disabled (figure 7.3a) and $n = 8$ for bounds enabled (figure 7.3b), as we reserve one bit for the sandboxing mechanism.

However, we ensure that adjacent allocations are always tagged with distinct tags and that memory is tagged with a new tag when it is freed, thus ensuring that spatial errors up to 16 bytes and temporal errors until the subsequent allocation of a chunk of memory are always caught.

$$V_{nr} = \frac{n!}{(n-k)!} = 240$$
$$V_t = n^k = 256$$
$$P(\text{c}) = 1 - \frac{V_{nr}}{V_t} = 6.25\%$$

(a) Probability of a tag collision with $n = 16$ and $k = 2$.

$$V_{nr} = \frac{n!}{(n-k)!} = 56$$
$$V_t = n^k = 64$$
$$P(\text{c}) = 1 - \frac{V_{nr}}{V_t} = 12.5\%$$

(b) Probability of a tag collision with $n = 8$ and $k = 2$.

Figure 7.3.: Probabilities of tag collisions for random tags.

7.5. MTE Performance evaluation

We evaluat the performance of different characteristics of MTE as implemented on the Tensor G3 chip. All the programs used to measure results in this section are implemented in Rust and available on GitHub[1]. As our benchmarking library, we use criterion[2]. After each benchmarking run, we let the device cool down for 30 seconds to prevent throttling. Additionally, we measure instruction throughput and latencies on all CPU cores.

7.5.1. Instruction Latencies and Throughput

We evaluate instruction latencies and throughput in microbenchmarks executing 100,000,000 iterations of 100 instructions to minimize the effect of the looping code. We use simpleperf[3] to measure CPU cycles of our benchmarking program[4]. For each instruction, we measure three variants: (1) the baseline, which includes the startup, setup, and teardown of the benchmark, where no actual instructions are executed, (2) the latency test, where we measure instructions with data dependencies between them, and (3) the throughput test, where we measure instructions with no data dependencies. We see the cycles and micro-ops per instruction in table 7.3.

We can make a few interesting observations:

- For the set-tag instructions, Cortex-X3 (out-of-order) has double the latency and half the throughput compared to Cortex-A720 (out-of-order).

[1] `https://github.com/martin-fink/mte-stg-bench`
[2] `https://github.com/bheisler/criterion.rs`
[3] `https://android.googlesource.com/platform/system/extras/+/master/simpleperf/doc/README.md`
[4] `https://github.com/martin-fink/mte-inst-cycles`

- On Cortex-A510 (in-order), `st2g` has double the latency and half the throughput of `stg`, while also being the only one with two micro-ops per instruction. This leads us to believe that this instruction performs the same micro-op as `stg`, but for two tag granules, while the operation is optimized on the other cores.
- Loading tags has a higher latency and lower or equal throughput than storing tags, except for the Cortex-X3, where both types of instructions require one cycle.
- As expected, we observe that latency-bound instructions perform as many or more cycles per instruction compared to throughput-bound instructions.

Table 7.3.: MTE cycles per instruction when latency- and throughput-bound (lower is better), and micro-ops per instruction.

Variant	Cortex-X3			Cortex-A720			Cortex-A510		
	Lat	Tp	μops	Lat	Tp	μops	Lat	Tp	μops
`irg`	2	0.75	3	2	1	3	3	2	1
`ldg`	1	1	2	1.5	1	2	4	4	1
`stg`	1	1	2	0.5	0.5	2	1.5	1	1
`st2g`	1	1	2	0.5	0.5	2	3	2	2
`stgp`	1	1	2	0.5	0.5	2	1.5	1	1
`stzg`	1	1	2	0.5	0.5	2	1.5	1	1

7.5.2. Tagging Primitives

We evaluate the performance of the different types of instructions to set the tag for a memory granule available in EL0 (user space) with the combinations described in table 7.4. Here, instruction refers to the instruction used to set the tag, and granule size refers to the amount of memory being tagged with a single instruction. The instructions `stzg` and `stgp` implicitly set the granule to zero, while we have to use an explicit memset for other instructions.

We run the benchmark on our testbed (see section 7.1) tagging a 128 MiB memory region. Before each run, we request a fresh piece of memory with `mmap` and run the specified configuration to prevent interference through already-filled caches.

We perform all runs on each type of CPU core on the Pixel 8 and illustrate the results in figure 7.4. As expected, the instructions implicitly setting the memory to zero are faster than tagging and then zeroing using an explicit memset. Both `stzg` and `stgp` are only slightly slower than a raw memset, as their memory accesses do not need to perform tag checks [3].

Table 7.4.: MTE benchmarking variants.

Variant	Instruction	Granule size	Implicit zero	memset
memset	-	-	No	Yes
stg	`stg`	16	No	No
st2g	`st2g`	32	No	No
stgp	`stgp`	16	Yes	No
stzg	`stzg`	16	Yes	No
stg+memset	`stg`	16	No	Yes
st2g+memset	`st2g`	32	No	Yes

Contrary to our expectations, `st2g` is only marginally faster than `stg`. This finding contradicts the data presented in section 7.5.1, where operations are performed with pre-filled data caches. In these benchmarks, we specifically measure wall clock time, not processor cycles, to evaluate performance when tagging a large region of memory immediately after requesting it from the operating system. This approach involves requesting a fresh segment of memory before each execution. The Cortex-X3 achieves shorter execution times than the Cortex-A720, despite us measuring lower instructions executed per cycle, due to its higher clock speed.

7.5.3. Synchronous and Asynchronous Mode

We evaluate the performance of sequential memory accesses with MTE disabled and enabled using synchronous mode and asynchronous mode on each type of CPU core on the Pixel 8. This represents the raw overhead of enabling MTE without additional instructions. In figure 7.5, we see that with synchronous mode, `memset` is 11.5%, 8.9%, and 13.2% slower on the respective cores compared to the baseline with MTE disabled. Asynchronous mode is closer to the baseline with an overhead of 0.9%, 3.7%, and 6.1% respectively.

7.5.4. Migrating Tagged Memory

Migrating tagged memory involves the coordinated transfer of both data and its associated tags. In figure 7.6, we analyze the performance of two migration strategies.

1. Baseline (`memcpy` with MTE): This baseline establishes the cost of a standard memory copy operation with MTE enabled.

2. MTE Disable/Re-enable: Disabling MTE, copying data with `memcpy`, transferring tags, and re-enabling MTE. This method temporarily compromises memory

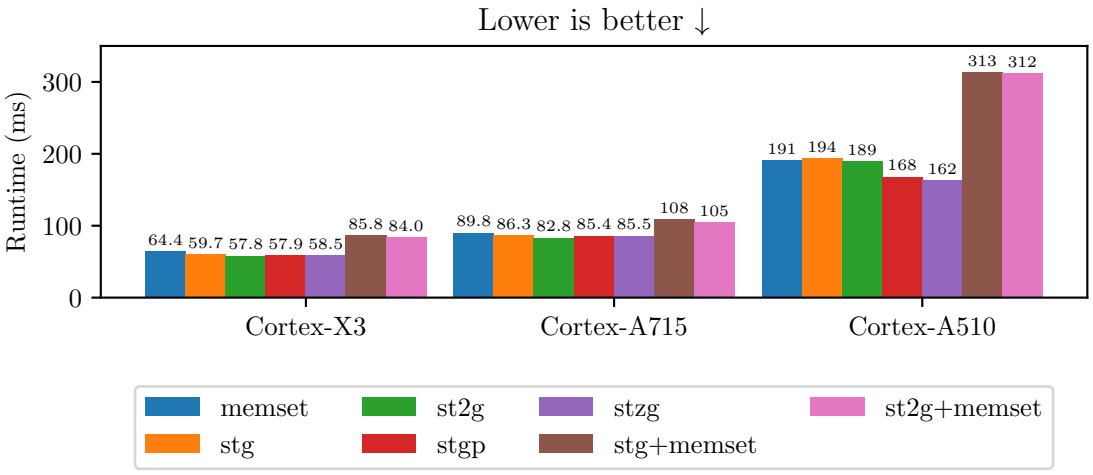


Figure 7.4.: Performance results of the benchmarking variants from table 7.4 on 128 MiB of memory.

safety during the transfer if other threads rely on MTE being active during this approach.

3. Iterative Copying: Copying tags and data in 16 byte granules allows copying tags and data simultaneously while keeping MTE active for other threads.

Iterative copying has an overhead of 9.4 %, 12.0 %, and 5.2 % compared to the baseline of copying just data on the Cortex-X3 (out-of-order), Cortex-A720 (out-of-order), and Cortex-A510 (in-order), respectively. Disabling and re-enabling MTE incurs an overhead of 5.25 %, 21.8 %, and 18.1 % respectively. This is in line with the findings in section 7.5.1, which shows that the Cortex-X3 core requires more cycles per tagging instruction compared to the Cortex-A720.

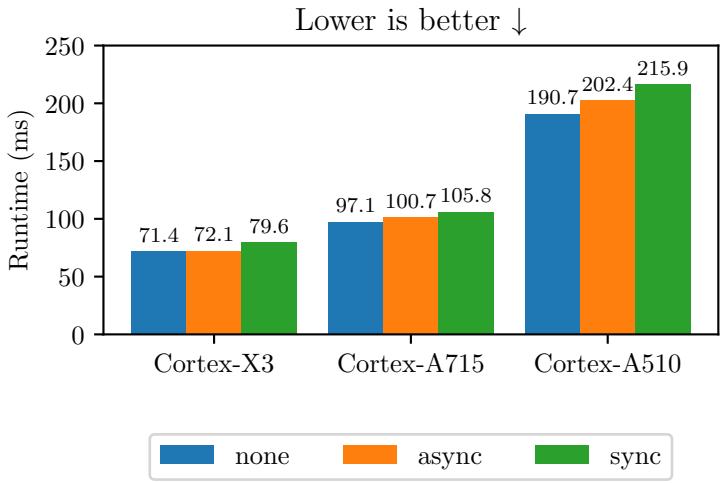


Figure 7.5.: Runtime of `memset` on 128 MiB of memory using different MTE modes.

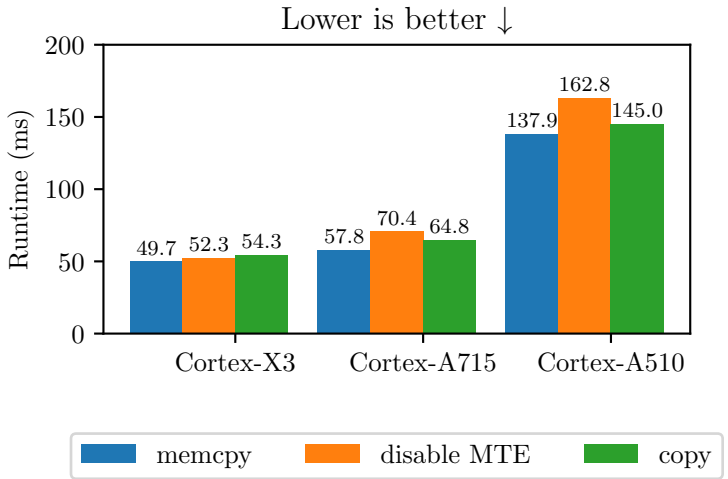


Figure 7.6.: Runtime comparison of strategies for moving 128 MiB of memory and tags between regions.

8. Related Work

This chapter reviews existing efforts in ensuring memory safety, focusing on WASM and C/C++ compiled directly to machine code. We explore notable projects in these areas, comparing their approaches with our research to highlight our distinct contributions.

8.1. Memory Safety for WebAssembly

This thesis builds upon existing efforts in the field of memory safety for WASM. Here, we examine notable projects aiming to achieve similar goals and highlight the distinct contributions of our research.

8.1.1. MS-WASM

A significant project in this domain is MS-WASM, a memory safety extension for WASM introduced by Disselkoen et al. and further developed by Michael et al. [8, 16]. MS-WASM introduces a new *segment memory* distinct from the linear memory, preventing accesses through arbitrary offsets. The segment memory relies on access through unforgeable *handles*, akin to CHERI pointers [38].

Key Differences This thesis adopts a different approach by enabling a gradual migration of memory segments into the linear memory. This preserves compatibility with unmodified code, as only the allocation of memory regions has to be changed. Memory accesses are still performed through integers, not pointers. We do not implement intra-object memory safety to enable an implementation with MTE. While MS-WASM proposes a technique to implement intra-object safety with a tagged-memory approach (shading), this is not supported by MTE. We, thus, choose this tradeoff, because utilizing MTE allows our implementation to run with significantly lower overhead on devices equipped with this hardware feature.

8.1.2. RichWasm

RichWasm is a richly typed intermediate language for safe memory interactions between languages with varying memory management models [21]. It allows for static

detection of memory safety violations and is especially beneficial for mixed-language interoperability.

RichWasm is intended as a compilation target for safe languages like Rust or OCaml, which have strong memory safety guarantees encoded in their type systems. Languages like C, which lack information for static type safety analysis, are not directly supported by RichWasm's type-driven safety model.

8.1.3. Pointer Authentication

Rehde [26] has worked on implementing pointer authentication primitives. In their work, they add pointer authentication primitives for data pointers backed by ARM's PAC to the memory safety extension described in this thesis. This complements our work by providing additional protection mechanisms against pointer corruption.

8.2. Memory Safety for C

Various projects exist that work on providing memory safety for C. In this section, we discuss several approaches to provide memory safety for C outside the context of WASM, such as language extensions (section 8.2.1), instrumentation-based methods (section 8.2.2), and hardened libraries (section 8.2.3).

8.2.1. Memory-Safe C Language Dialects

Several projects, such as CCured [19] or Cyclone [11], implement memory safety by extending the C language. Their approach adds information about allocations but requires manual changes to existing code. CHERI C/C++ [35] goes in a similar direction, widening pointers to 128 bits. CHERI C/C++ can be compiled in one of two modes: (1) purecap mode, where all pointers are capabilities and the compiler restricts them to allocations, automatically providing memory safety, and (2) hybrid mode, where pointers remain 64 bits wide. Programmers can annotate pointers, transforming them into capabilities, allowing to mix capability-aware and unmodified code. These approaches break binary compatibility by storing metadata in a fat pointer representation.

8.2.2. Instrumentation-Based Memory Safety

The following projects automate memory safety without source code changes, employing various strategies to detect and prevent memory errors [28, 29, 20]. We can divide these approaches into three major categories: (1) trip-wire-based, (2) object-based, and (3) pointer-based.

Trip-Wire-Based Approach

Trip-wire-based projects guard zones around allocations to catch memory safety errors. Some projects that fall into this category are ASan [28], GWP-ASan [29], SafePM [5], or Valgrind Memcheck [20]. In the case of ASan, 1/8 of the whole address space is mapped as shadow memory, with one bit of shadow memory representing the state of one byte of memory. When allocating memory, the allocation is padded, and guard zones around the allocation are marked as inaccessible. When freeing memory, the whole memory block is marked as inaccessible. The sanitizer inserts checks before each memory access that check the address being accessed. ASan was built as a debugging tool to be used when running tests or fuzzing programs, as it has an average overhead of 73 %. SafePM [5] utilizes ASan to provide the same memory safety guarantees for applications utilizing persistent memory.

GWP-ASan [29] is intended to be deployed in production. It adopts a probabilistic approach to memory safety by sampling a subset of allocations and placing them in a guarded memory region to detect spatial and temporal memory safety violations for those allocations. Its low selection probability minimizes runtime overhead. The goal is to uncover hard-to-reproduce memory bugs triggered by real-world user behavior not covered by fuzzing or tests.

In contrast, Valgrind Memcheck utilizes dynamic binary instrumentation, which does not require recompilation from source, but incurs much larger overheads.

Object-Based Approach

For object-based approaches, the metadata is associated with the allocated object. A notable project in this area is Baggy Bounds Checking [2], which aligns and pads allocations to powers of two to speed up bounds checks at runtime. Metadata about the allocation can be efficiently retrieved thanks to the alignment of the allocation. Usually, pointers are checked when doing pointer arithmetic, not when dereferencing pointers. This poses a challenge for C, as the language permits out-of-bounds pointers as long as they are not dereferenced. Intra-object safety is more challenging for object-based approaches, as enforcing those additional constraints may require more sophisticated data structures. For instance, Baggy Bounds Checking does not support intra-object safety.

Pointer-Based Approach

In contrast, pointer-based approaches keep track of the pointer bounds. These can be either stored in the pointer, as fat pointers [35] or in unused bits [30] or in an external data structure [18]. One project in this domain is SoftBound [18]. It keeps track of

each pointer's upper and lower bounds, instrumenting memory accesses to check if the pointer is within its bounds. This allows the implementation of intra-object memory safety, as multiple pointers with different bounds may point to overlapping regions. Additionally, creating out-of-bounds pointers is not an issue, as pointer arithmetic is not checked. SoftBound keeps track of pointers stored in memory in a separate data structure, where bounds are looked up and stored when loading and storing pointers from and to memory. Similarly, the bounds must be propagated as additional arguments and return values when passing pointers to and from functions.

CHERI [38] implements a pointer-based approach with hardware support. It extends pointers to 128 bits, including permission and compressed bounds. Pointer dereferences are checked by hardware, promising much better performance than software-based checks. Currently, hardware exists in the form of the ARM Morello board [36], a limited-production development board shipped to selected academic and industry partners.

Hybrid Approaches

We consider memory-tagging-based approaches as a hybrid of object- and pointer-based approaches. HWASan [30] or MTE both associate metadata with the pointer in its upper, unused bits and with objects by assigning tags to memory. In the case of HWASan, only memory accesses are instrumented, while memory accesses are checked by hardware with MTE. This allows out-of-bounds pointers to exist if they are not dereferenced. However, intra-object safety is not possible to implement, similar to object-based approaches. In the case of both HWASan and MTE, binary compatibility is preserved, as the hardware ignores extra metadata in the upper bits of pointers, thus allowing uninstrumented code to handle pointers with and without metadata.

8.2.3. Hardened Memory Allocators

A few memory allocators have implemented support for MTE. They provide probabilistic memory safety against both spatial and temporal memory safety as long as no tag collisions occur.

- Scudo Hardened Allocator (used in Android): A security-oriented allocator providing defense mechanisms against heap-based vulnerabilities [27].
- Chrome's PartitionAlloc: A partitioning allocator focusing on security and efficiency for multithreaded environments [6].
- glibc's Ptmalloc: The GNU C library's standard memory allocator, with evolving experimental support for MTE [9].

The Cling memory allocator [1] uses a different approach to prevent use-after-free exploits by placing heap metadata out-of-band and reusing memory only for objects of the same type. It achieves this by analyzing the call stack to determine the type of data being allocated, and it works as a drop-in replacement without any code changes.

9. Conclusion

In this thesis, we present three pieces of work: (1) a minimally-invasive and adaptable WASM extension that provides memory safety primitives to compilers and programmers, (2) an implementation consisting of (2a) a compiler toolchain integrated into LLVM, including a modified wasi-libc and an allocator to provide spatial and temporal memory safety for the heap, an LLVM sanitizer pass to instrument stack allocations, (2b) an implementation in wasmtime, compiling and running the WASM extension on MTE hardware, and utilizing MTE as a replacement for software-based bounds checks, and (3) an evaluation of our work and a performance analysis of MTE, performed on real hardware.

9.1. Future Work

9.1.1. Additional Implementations

We implement our prototype based on MTE for our memory safety extension. However, additional implementations exploring different extensions, such as ARM's TBI, available from ArmV8.0, would allow storing metadata in the top byte while performing access checks in software, similar to HWASan [30]. Software-based implementations, while slower, would allow deploying the memory safety extension to more devices without support for MTE.

We are working on an implementation utilizing CHERI, with an in-progress implementation for the ARM Morello development board [36]. The CHERI architecture provides much more fine-grained checks and unlimited compartments. However, it requires widening pointers to 128 bits and moving from a fixed 16 byte alignment for segments to a dynamic alignment, depending on the segment size. Additionally, revoking capabilities for temporal memory safety is more complicated than MTE [39], where memory can be retagged. Exploring and comparing these tradeoffs will be part of our future work.

9.1.2. Backward Compatibility

Currently, code compiled with the memory safety extension requires a modified runtime aware of this extension. We are exploring techniques to embed metadata about allocations in custom WASM segments that are ignored by runtimes but are used to provide memory safety when running on a modified runtime.

9.1.3. Combining Guard Pages and MTE

Currently, we are limiting the number of sandboxes to 15, as we allocate the zero tag for the runtime and one tag per instance. Future work might explore possibilities to increase the number of sandboxes by combining MTE with guard pages.

9.1.4. Pointer Authentication

In a previous Bachelor's thesis, Rehde explored the integration of pointer authentication primitives for data pointers to the WASM extension, with an implementation using ARM's PAC extension [26]. While WASM lacks raw function pointers, table indices remain vulnerable to overwriting and forgery. As we do not support intra-object safety, some overflow exploits remain possible. These could, for instance, target an object's virtual function table to redirect control flow to a different function.

Adding support for data pointers to sign and authenticate these table indices would provide another defense against such attacks.

A. Artifacts

The artifacts are available on GitHub. They consist of the following repositories:

- LLVM: `https://github.com/TUM-DSE/llvm-memsafe-wasm`
- wasmtime: `https://github.com/TUM-DSE/wasmtime-mte`
- wasm-tools: `https://github.com/TUM-DSE/wasm-tools-mte`
- wasi-sdk: `https://github.com/martin-fink/wasi-sdk`
- wasi-libc: `https://github.com/martin-fink/wasi-libc`
- PolyBench/C: `https://github.com/martin-fink/polybench-c`

A.1. Building

All the projects contain a `flake.nix` that can be used to start a development shell containing all the dependencies. They need to be installed manually on systems without Nix.

A.1.1. LLVM Toolchain

The following commands can be used to bootstrap the WASM compiler toolchain, including the libc.

```
git clone --recurse-submodules https://github.com/martin-fink/wasi-sdk.git
cd wasi-sdk
make package
cd dist
tar -xzf wasi-sdk-20.38g2b3e8f68d320-linux.tar.gz
tar -xzf libclang_rt.builtins-wasm64-wasi-20.38g2b3e8f68d320.tar.gz
mv lib wasi-sdk-20.38g2b3e8f68d320/lib/clang/18/
```

A.1.2. Wasmtime

We are building wasmtime for Android, as the Pixel 8 is the only device supporting MTE at the time of writing. The same steps are also possible if you are building for Linux. In any case, you need a C compiler and linker for the corresponding target.

```
git clone --recurse-submodules \
    https://github.com/TUM-DSE/wasmtime-mte.git wasmtime
cd wasmtime
```

Building for Android

You need to install the Android NDK[1] to compile wasmtime for Android. The resulting binary will be placed in `./target/aarch64-linux-android/release/wasmtime`.

```
rustup target add aarch64-linux-android
mkdir .cargo
echo << EOF
[env]
[target.aarch64-linux-android]
linker = "/path/to/android/aarch64-linux-android34-clang"
ar = "/path/to/android/llvm-ar"
rustflags = ["-C", "target-feature=+mte"]
EOF >> .cargo/config.toml
cargo build --release --target aarch64-linux-android
```

Building for Linux

You need a cross-compiler for aarch64 Linux. The resulting binary will be placed in `./target/aarch64-unknown-linux-gnu/release/wasmtime`.

```
rustup target add aarch64-unknown-linux-gnu
mkdir .cargo
echo << EOF
[env]
CC_aarch64-unknown-linux-gnu = "aarch64-linux-gnu-gcc"
CC_aarch64-unknown-linux-musl = "aarch64-linux-gnu-gcc"

[target.aarch64-unknown-linux-gnu]
```

[1] `https://developer.android.com/ndk`

```
linker = "aarch64-linux-gnu-gcc"
rustflags = ["-C", "target-feature=+mte"]

[target.aarch64-unknown-linux-musl]
linker = "aarch64-linux-gnu-gcc"
rustflags = ["-C", "target-feature=+mte"]
EOF >> .cargo/config.toml
cargo build --release --target aarch64-unknown-linux-gnu
```

A.2. Running Programs

A.2.1. Compiling with Memory Safety

LLVM supports the flags described in table 6.1.

```
WASI_SDK=wasi-sdk-20.38g2b3e8f68d320
"./$WASI_SDK/bin/clang" \
  -mmem-safety \
  -Os \
  -fsanitize=wasm-memsafety \
  --sysroot "$WASI_SDK/share/wasi-sysroot" \
  main.c \
  -o main.wasm
```

A.2.2. Running with Wasmtime

Copy the wasmtime binary built in appendix A.1.2 to an MTE-capable device, e.g., QEMU. Wasmtime supports the flags described in table 6.2.

```
./wasmtime run \
  -W memory64 \
  -W mem-safety \
  -C mte-bounds-checks \
  -C mte \
  main.wasm
```

Abbreviations

ASan Address Sanitizer

CHERI Capability Hardware Enhanced RISC Instructions

CLIF Cranelift IR

GEP `getelementptr`

HWASan Hardware-Assisted Address Sanitizer

IR intermediate representation

ISA instruction set architecture

MMU Memory Management Unit

MTE Memory Tagging Extension

PAC Pointer Authentication

rss resident set size

TBI Top Byte Ignore

WASI WebAssembly System Interface

WASM WebAssembly

List of Figures

List of Tables

Bibliography

[1] P. Akritidis et al. "Cling: A memory allocator to mitigate dangling pointers." In: *19th USENIX Security Symposium (USENIX Security 10)*. 2010.

[2] P. Akritidis, M. Costa, M. Castro, and S. Hand. "Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors." In: *USENIX Security Symposium*. Vol. 10. 2009, p. 96.

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

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

[5] K. K. Bozdoğan, D. Stavrakakis, S. Issa, and P. Bhatotia. "SafePM: A sanitizer for persistent memory." In: *Proceedings of the Seventeenth European Conference on Computer Systems*. 2022, pp. 506–524.

[6] *Chromium PartitionAlloc*. Accessed on March 28, 2024. URL: https://chromium.googlesource.com/chromium/src/+/master/base/allocator/partition_allocator/.

[7] *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 (visited on 04/19/2015).

[8] C. Disselkoen, J. Renner, C. Watt, T. Garfinkel, A. Levy, and D. Stefan. "Position paper: Progressive memory safety for webassembly." In: *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. 2019, pp. 1–8.

[9] *glibc ptmalloc*. Accessed on March 28, 2024. URL: https://ftp.gnu.org/gnu/glibc/.

[10] 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.

[11] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. "Cyclone: a safe dialect of C." In: *USENIX Annual Technical Conference, General Track*. 2002, pp. 275–288.

[12] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. "Spectre attacks: Exploiting speculative execution." In: *Communications of the ACM* 63.7 (2020), pp. 93–101.

[13] C. Lattner and V. Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." In: *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[14] 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.

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

[16] A. E. Michael, A. Gollamudi, J. Bosamiya, E. Johnson, A. Denlinger, C. Disselkoen, C. Watt, B. Parno, M. Patrignani, M. Vassena, et al. "Mswasm: Soundly enforcing memory-safe execution of unsafe code." In: *Proceedings of the ACM on Programming Languages* 7.POPL (2023), pp. 425–454.

[17] M. Musch, C. Wressnegger, M. Johns, and K. Rieck. "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. 2019, pp. 23–42.

[18] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. "SoftBound: Highly compatible and complete spatial memory safety for C." In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2009, pp. 245–258.

[19] G. C. Necula, S. McPeak, and W. Weimer. "CCured: Type-safe retrofitting of legacy code." In: *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 2002, pp. 128–139.

[20] N. Nethercote and J. Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation." In: *ACM Sigplan notices* 42.6 (2007), pp. 89–100.

[21] Z. Paraskevopoulou, M. Fitzgibbons, M. Thalakottur, N. Mushtak, J. S. Mazur, and A. Ahmed. "RichWasm: Bringing Safe, Fine-Grained, Shared-Memory Interoperability Down to WebAssembly." In: *arXiv preprint arXiv:2401.08287* (2024).

[22] B. C. Pierce. *Types and programming languages*. MIT press, 2002.

[23] G. D. Plotkin. "A structural approach to operational semantics." In: (1981).

[24] L.-N. Pouchet. *Polybench: The polyhedral benchmark suite*. Accessed: 2024-03-25. URL: https://web.cs.ucla.edu/~pouchet/software/polybench/.

[25] 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.

[26] F. Rehde. "Hardware-Assisted Memory Safety for WebAssembly." BA thesis. Technical University of Munich, 2023.

[27] *Scudo Hardened Allocator*. Accessed on March 28, 2024. URL: https://llvm.org/docs/ScudoHardenedAllocator.html.

[28] 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.

[29] 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).

[30] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov. "Memory Tagging and how it improves C/C++ memory safety." In: *arXiv preprint arXiv:1802.09517* (2018).

[31] 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.

[32] R. Szewczyk, K. Stonehouse, A. Barbalace, and T. Spink. "Leaps and bounds: Analyzing WebAssembly's performance with a focus on bounds checking." In: *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2022, pp. 256–268.

[33] 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).

[34] 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).

[35] R. N. M. Watson, A. Richardson, B. Davis, J. Baldwin, D. Chisnall, J. Clarke, N. Filardo, S. W. Moore, E. Napierala, P. Sewell, and P. G. Neumann. "CHERI C/C++ Programming Guide." en. In: (June 2020).

[36] R. N. M. Watson, G. Barnes, J. Clarke, R. Grisenthwaite, P. Sewell, S. W. Moore, and J. Woodruff. *Arm Morello Programme: Architectural security goals and known limitations*. Tech. rep. UCAM-CL-TR-982. University of Cambridge, Computer Laboratory, July 2023. DOI: 10.48456/tr-982. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-982.pdf.

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

[38] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. "The CHERI capability model: Revisiting RISC in an age of risk." In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 457–468.

[39] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. Watson, et al. "Cherivoke: Characterising pointer revocation using cheri capabilities for temporal memory safety." In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 545–557.