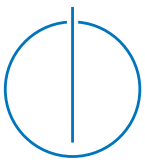
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Hardware-Assisted Memory Safety for WebAssembly

Fritz Rehde



SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

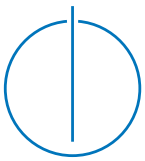
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Hardware-Assisted Memory Safety for WebAssembly

Hardware-unterstützende Speichersicherheit für WebAssembly

Author:	Fritz Rehde
Supervisor:	Prof. Dr. Pramod Bhatotia
Advisor:	Martin Fink
Submission Date:	August 15, 2023



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

Munich, August 15, 2023 Fritz Rehde

Acknowledgments

I would like to extend my thanks to the following individuals for their contributions to this thesis:

To my supervisor, Prof. Pramod Bhatotia, for offering me the chance to work on such an interesting project with diverse real-world implications and significance. To my advisor Martin Fink, who has been an invaluable mentor throughout this journey. His unwavering support, guidance, and patience were pivotal. Martin's willingness to answer any questions and share his expertise in compiler development, as well as his thorough evaluation and feedback of my pull requests and contributions, have greatly enriched my learning experience.

Abstract

With the rise of ARM hardware extensions offering advanced safety and performance features, the potential of WebAssembly (WASM) as a portable, efficient, and safe computing platform has grown. In this project, we try to enhance WebAssembly by creating a functional prototype that consists of extensions to both LLVM and a chosen WASM runtime called Wasmtime. This allows WebAssembly to leverage new ARM64 hardware extensions, namely the Memory Tagging Extension (MTE) and Pointer Authentication (PAC), to enhance memory safety for WebAssembly modules in their sandboxed environments. Through this integration, our toolchain is able to detect spatial and temporal memory safety issues at runtime in WASM applications derived from vulnerable C programs. Notably, our approach requires no alterations to the input C source code, since only the toolchain has been adapted.

Although many of our optimization strategies show promise in our performance benchmarks, there are observable runtime performance overheads when operating within the emulated QEMU environment. The use of QEMU was a necessary compromise given the unavailability of some hardware extensions in current devices. We anticipate that evaluations on actual hardware, once available, will provide a more accurate representation of our toolchain's efficiency.

Zusammenfassung

Durch die fortschreitende Entwicklung von ARM-Hardware-Erweiterungen mit optimierten Sicherheits- und Leistungsmerkmalen steigt das Potenzial von WebAssembly (WASM) als portable, effiziente und sichere Computing-Plattform. In diesem Projekt haben wir versucht, WebAssembly durch die Erstellung eines funktionsfähigen Prototypen, welcher Erweiterungen für LLVM und eine ausgewählte WASM-Laufzeit namens Wasmtime beinhaltet, weiterzuentwickeln. So kann WebAssembly die neuen ARM64-Hardware-Erweiterungen, insbesondere die Memory Tagging Extension (MTE) und Pointer Authentication (PAC), nutzen, um die Speichersicherheit für WebAssembly-Module in ihren isolierten Umgebungen zu stärken. Dank dieser Integration ist unsere Toolchain in der Lage, räumliche und zeitliche Speichersicherheitsprobleme in WASM-Anwendungen zu erkennen, die von anfälligen C-Programmen stammen. Es ist besonders hervorzuheben, dass unser Ansatz keine Änderungen am ursprünglichen C-Quellcode benötigt, da ausschließlich die Toolchain modifiziert wurde.

Obwohl viele unserer Optimierungsansätze in den Leistungstests vielversprechend erscheinen, konnten wir deutliche Leistungseinbußen im emulierten QEMU-Umfeld feststellen. Die Verwendung von QEMU war notwendig, da einige der Hardware-Erweiterungen in aktuellen Geräten noch nicht verfügbar sind. Wir erwarten, dass Tests mit echter Hardware, sobald verfügbar, ein genaueres Bild von der Effizienz unserer Toolchain bieten werden.

Contents

1 Introduction

1.1 Motivation

In the evolving landscape of software development, memory safety issues, such as buffer overflows and use-after-free bugs, continue to be a pivotal issue. Especially in applications developed using programming languages such as C, these vulnerabilities remain present due to the inherent lack of memory safety guarantees. The ongoing prevalence of these challenges, even amidst significant strides in programming paradigms, underlines the pressing need for practical and effective mitigation measures.

WebAssembly (WASM) is a binary instruction format that has quickly gained prominence, starting as a promising alternative and supplement to JavaScript, but soon expanding its horizon far beyond the confines of web applications. WASM offers a unique blend of portability, performance and security across various platforms, serving as a compilation target from a variety of programming languages, including C, C++ and Rust. A crucial safety feature of WASM is its ability to form a secure sandbox that shields the hosting system, be it a browser or an operating system, from the WASM guest code executed within it. However, while this ensures the host's protection from potentially harmful WASM programs, it does not shield the WASM program from its own vulnerabilities. This means that WASM modules, especially when compiled from inherently vulnerable languages, remain susceptible to memory safety threats.

Recent breakthroughs in the realm of hardware engineering have brought forward some promising solutions. ARM's Memory Tagging Extension (MTE) and Pointer Authentication Code (PAC) emerge as innovative techniques designed to enhance memory protection directly at the hardware level while ensuring minimal performance overheads. While these methods bring great promise, they are not standalone solutions. They offer interfaces to essential memory safety capabilities, but a separate challenge emerges: seamlessly integrating these hardware features into existing software landscapes, such as compilers, runtimes, and notably, platforms like WebAssembly.

1.2 Contributions

Our study seeks to address a distinct research gap, namely the integration of WebAssembly with ARM64 Hardware Extensions. The pivotal question driving this thesis is: How can we utilize ARM's hardware extensions to provide memory safety in WebAssembly without incurring significant performance overhead?

In pursuit of a solution, this work strives to merge ARM's MTE and PAC functionalities into WebAssembly's framework. Our endeavors extend beyond theoretical concepts, as we propose and implement practical extensions to the WebAssembly standard with a special focus on C language integration using the WASI-libc. By incorporating changes into the LLVM ecosystem [1] and the Wasmtime runtime [2], we pave the way for a more secure WebAssembly, equipped with advanced hardware-based memory protection. Along this path, we introduce refined concepts like segments, tagged indexes, and a new set of WebAssembly instructions, all designed to leverage the capabilities of MTE and PAC fully.

Central to our approach is also a commitment to usability. A highlight of our contributions is the seamless integration into the LLVM compiler and WASM infrastructure, ensuring that the source code of input C programs can remain unchanged, ensuring that developers can easily integrate these enhancements without overhauling their existing codebase.

Our contributions include optimizing memory tagging algorithms, integrating new ARM hardware extension instructions into Wasmtime with enhanced error messaging, introducing LLVM analysis passes for Pointer Authentication, and expanding memory handling to encompass `calloc` and `realloc`. Additionally, we devised a strategy to tag the linear heap, thereby optimizing Wasmtime to eliminate the need for bounds checks.

To assess our contributions, we put our toolchain through a variety of performance benchmarks. This assessment serves a dual purpose: analyzing the strengths of our implementations and identifying opportunities for further improvements.

Our efforts contribute to the ongoing evolution of WebAssembly. By enhancing its security and capabilities, we aim to support WebAssembly's growing reputation as a platform for secure, performant, and portable computing across diverse platforms and applications.

[1] `https://github.com/TUM-DSE/llvm-memsafe-wasm`
[2] `https://github.com/TUM-DSE/wasmtime-mte`

2 Background

In this chapter we will discuss the technical background required for the later sections. We will define the term *memory safety* in Section 2.1, the WebAssembly binary instruction format in Section 2.2, the LLVM compiler toolchain in Section 2.3, and the ARM64 processor architecture in Section 2.4. Furthermore, we will highlight some of WebAssembly's key limitations in 2.2.2, which will underline the motivation for and necessity of our project.

2.1 Memory Safety

We can define a memory safe program in the following way:

Definition 1 *A program is memory safe if all memory pointers refer to valid memory when being dereferenced.*

While this definition is intuitive, it is quite vague, lacks formal precision, and falls into the trap of defining memory safety as simply the lack of memory-related errors.[1]

This challenge in formalizing a definition can be attributed to our reliance on informal, everyday intuitions when discussing memory safety [1]. In practice, many memory management-related issues can be categorized as either *spatial* or *temporal* memory safety violations [13]. *Buffer overflows*, which occur when data is accessed outside the bounds of an array, represent spatial memory errors. On the other hand, *use-after-free* and *double-free* errors, which involve memory being accessed or freed after it has been already freed, are temporal memory safety violations. Other issues such as *memory leaks*, where the programmer forgets to free allocated memory leading to memory exhaustion, and the use of uninitialized memory, are generally also classified as temporal errors.

Furthermore, many real-world C programs intentionally *rely on* memory management techniques, such as unrestricted pointer arithmetic, that can become memory-related issues if used incorrectly [1]. Some argue that these memory management-related issues should be classified as errors, and a programming language specification should include expected behavior for when these errors are encountered [1]. In contrast, specifications like the C programming language standard [22] treat these issues as undefined behavior, leaving their handling to the discretion of compiler implementations.

From a security standpoint, the critical issue here is not the occurrence of the errors themselves, but the fact that they lead to undefined behavior. When undefined behavior ensues, the program's behavior is influenced by complicated, low-level factors, like how the compiler arranges memory at runtime. These factors often result in exploitable vulnerabilities. On the other hand, programs written in memory-safe languages like Java may still contain the aforementioned memory-related issues, but such errors lead to sensible and predictable outcomes, such as thrown exceptions.[1]

The significance of memory safety issues is underlined by large technology companies, such as Google [46] and Microsoft [32], reporting that around 70 percent of the vulnerabilities found in their software products were the result of memory safety issues. Memory safety's importance is further emphasized by the fact that the National Security Agency of the United States of America has published a Cybersecurity Information Sheet [36] on the topic.

Even though we have argued that our initial memory safety definition 1 has its limitations, we will nonetheless use it in this paper.

2.2 WebAssembly

WebAssembly (WASM) [20] is a portable, low-level bytecode format collaboratively developed by major browser vendors Google, Mozilla, Microsoft, and Apple. It was initially envisioned as an efficient and secure replacement for, or alternative to, JavaScript, which, until then, had been the Web's only built-in language. However, WebAssembly was also designed for potential applications beyond the Web. Its nature as an "abstraction over modern hardware" [20] makes it language-, hardware-, and platform-independent, and, thereby, an attractive compilation target for general applications.

2.2.1 Stack-based Virtual Machine

The official WebAssembly documentation describes WebAssembly as "a binary instruction format for a stack-based virtual machine" [48]. To dissect this definition, and understand how it enables WebAssembly to support the *compile once, run everywhere* paradigm, it is helpful to understand the terms *instruction set architecture (ISA)*, *binary instruction format*, *virtual machine* and *stack-based*.

Virtual Instruction Set Architecture An instruction set architecture (ISA) defines the supported data types, the registers, the programming model, the memory architecture, the instruction format, and the addressing modes, among other characteristics [37]. In the case of WebAssembly, this ISA is virtual, meaning it defines an abstract machine

model and instruction set that WebAssembly programs are designed for, but does not correspond to any specific real-world hardware.

Binary Instruction Format The binary instruction format [49] represents a low-level encoding of the machine instructions defined by the ISA. This format, which the WebAssembly virtual machine can directly execute, is essential for the portability and compactness of WebAssembly. It enables efficient transmission over the internet and fast decoding and execution. On the other hand, for the convenience of developers, WebAssembly also supports a human-readable textual format [50]. This format facilitates manual writing, reading, and understanding of WebAssembly code, and is typically used in development and debugging contexts [34].

Virtual Machines WebAssembly provides an abstract virtual machine model that WebAssembly programs are designed for [20]. In the context of WebAssembly, the term *virtual machine* does not refer to a full-scale *system virtual machine*, which simulates a complete hardware system and runs an entire operating system on top. Instead, WebAssembly is designed for a *process virtual machine*, also known as an *application virtual machine*. This VM is tailored to run a single program, which may itself be multi-threaded or multi-process, and isolate it from the host system for security, portability, or abstraction purposes. The abstract machine does not correspond to any specific real-world hardware but is an idealized, platform-agnostic model that can be implemented on various real-world hardware platforms. As a result, WebAssembly code can be compiled once and run anywhere on any system that provides a WebAssembly VM implementation. Furthermore, WebAssembly is designed to be lightweight and efficient, typically implemented as a library within the host environment. [20]

Abstract Computing Machines Stack-based and register-based virtual machines represent two distinct types of abstract computing machines [23] used to execute instructions.

A stack-based virtual machine operates on a stack data structure, with most operations involving popping values from the stack, performing the operation, and pushing the result back onto the stack. This design leads to simple instruction sets because the location of operands, at the top of the stack, is implicitly known.

Contrarily, a register-based virtual machine uses a model similar to most physical CPU architectures. Despite having more complex instruction sets, because each instruction needs to explicitly specify the registers it operates on, these VMs can offer performance advantages. They can perform operations on any pair of registers directly, which eliminates the need for pushing and popping data from a stack. Addition-

ally, their bytecode is often denser, because they can accomplish the same tasks as stack-based VMs with fewer instructions. For example, simple arithmetic operations typically require a single instruction in register-based VMs, but multiple instructions in stack-based VMs to handle the necessary stack operations.

As mentioned previously, WebAssembly is based on a stack-machine model. Thanks to its strict type system, WebAssembly can statically determine the layout of the operand stack ahead of time at any given point in the code [20]. Simply put, a strict type system means that the type of data on the stack can be known before the code is executed.

This directly represents one of the advantages of stack-machine models, which is that it is simple to verify their types and safety. This is critical for WebAssembly's use on the Web, where it is essential to efficiently verify that a program is safe to run. Furthermore, Shi et. al [43] has shown that stack-based models result in a smaller footprint than register machines, which is advantageous for the efficient transmission of code over the internet. Finally, stack-based machines offer greater portability, since they do not depend on the specific number and type of registers in the underlying physical architecture.

However, actual implementations can also use this static determination of the operand stack to optimize how the data flows from one instruction to another without having to physically create and maintain the operand stack. In practice, this means real-world WebAssembly implementations (such as Wasmtime, which is discussed in Section 2.2.5) do not necessarily interpret the stack-based instructions one by one, like a simple stack machine would. Instead, they typically use Just-In-Time (JIT) compilation to translate WebAssembly code into native machine code for the host hardware. This generated machine code often uses a register-based model, like most real CPUs do, and optimizes the data flow between instructions to avoid unnecessary stack operations.

In summary, while WebAssembly appears as a stack machine on the surface, its stack-based structure is largely an abstraction. This highlights an interesting aspect of WebAssembly, namely that its design allows it to reap the benefits of both stack-based and register-based machines.

2.2.2 Linear Memory

In traditional computing environments, a mechanism called the *virtual address space* is commonly used to manage memory. To understand the unique approach of WebAssembly's memory model, it is essential to first comprehend the functionality and limitations of this traditional method.

Virtual Address Space In computing, virtual addressing [31] is a memory management technique widely employed by modern operating systems. It allows a program

to view memory as a contiguous, logical address space, regardless of the underlying physical memory layout. This virtual address space is usually much larger than the actual physical memory of the system, and it provides a consistent and standardized environment for programs to operate in. There are several advantages to this approach. Firstly, it abstracts away the complexities of physical memory management, allowing programs to operate as if they have access to a large, uniform block of memory. This simplifies programming, as developers do not need to worry about the details of memory allocation and can focus on their application logic. Virtual addressing also improves the isolation between different programs running on the same system, enhancing system stability and security. Each program operates within its own virtual address space, so it cannot accidentally or intentionally interfere with the memory of another program. However, differences in how operating systems and hardware handle virtual addressing, such as differences in address space size, can introduce complexity and portability issues. This variation can lead to compatibility problems when trying to run the same program on different systems.

On the other hand, WebAssembly introduces the concept of *linear memory*. At its most fundamental level, WebAssembly's linear memory is a contiguous, growable array of bytes, which can be read from and written to by a WebAssembly module. Every byte in this array can be accessed via an *index*, which starts from 0 and goes up to the size of the memory minus one, and can be seen as an offset into the array. This memory model abstracts away the complexities of managing different segments of memory. Each WebAssembly module *instance* [20], which is the dynamic representation of a module complete with mutable memory and an execution stack, can have up to one associated linear memory. This is because some modules, perhaps those only performing calculations using its local variables and parameters, do not need to have a linear memory at all.

There are several reasons for WebAssembly's choice to use linear memory instead of providing direct access to the virtual address space. Firstly, the linear memory model is more portable across different platforms. While virtual address spaces can have different properties on different operating systems or hardware platforms, a simple array of bytes behaves consistently. This consistent behavior is crucial for WebAssembly's goal of providing a universal binary format. Furthermore, the simplicity of the linear memory model significantly eases the process of compiling programs from programming languages that utilize direct memory access, like C/C++, into WebAssembly. Many such programs rely on pointer arithmetic for data manipulation, which can be smoothly translated into WebAssembly memory operations as simple integer offsets in the linear memory array (*indexes*).

However, the linear memory model is not without its downsides. It introduces an additional layer of indirection, which can impact performance. Also, the simplicity of a

single, flat array can limit the flexibility for more complex memory management tasks.

Sandboxing In the context of safety and security, the linear memory model also offers a unique advantage through its *sandboxing* feature [20]. Linear memory is sandboxed and isolated for each WebAssembly module instance. This isolation means that a WebAssembly module cannot accidentally or maliciously interfere with the memory of the host system or other modules, which is a significant concern when running untrusted code from the web. To enforce this sandboxing, WebAssembly employs *bounds checking*. Essentially, every time a WebAssembly module tries to access a particular memory location via an index, a check is performed to verify that the index falls within the current size of the linear memory. If the check fails, the access is denied and an out-of-bounds error is triggered. This way, the memory sandbox cannot be violated. However, while the sandboxing mechanism preserves the safety of the host environment, it's crucial to note that WebAssembly does not inherently offer memory safety, which we defined earlier in Section 2.1, *within* the sandbox. The linear memory is essentially a contiguous array of bytes, and WebAssembly does not enforce any rules on how this memory should be used. This means that programs compiled to WebAssembly are as safe, or unsafe, as they were in their source language. For instance, if a program written in C, a language notorious for its lack of memory safety, is compiled to WebAssembly, it can still suffer from the same kinds of memory safety issues in WebAssembly as it could in its native environment. This highlights an important aspect of WebAssembly's sandboxing security model: while it protects the host environment from the WebAssembly module, it doesn't protect the module from itself. This also sets the stage well for our project, which aims to bring memory safety to WebAssembly.

To summarize, the linear memory model used by WebAssembly is another abstraction that helps to achieve its goals of safety, portability, and efficient execution. It simplifies memory management, improves security, and facilitates the compilation of other languages to WebAssembly. In practical implementations, WebAssembly's linear memory is generally realized through a dedicated region of the virtual address space of the host system. This arrangement allows efficient translation of memory access operations within the sandboxed environment to the underlying system's memory, retaining the performance benefits of direct memory access while preserving the safety boundaries of the sandbox.

2.2.3 Supported Environments

Basic Data Types WebAssembly's type system, at its core, only includes four basic value types: integers and IEEE 754 [21] floating point numbers, each in 32 and 64 bit

width [20]. The choice of these types is a reflection of the fact that these are the data types natively supported by CPUs and can be efficiently manipulated. Interestingly, the 32-bit restriction in WebAssembly is specific to the index of the linear memory, not the data types themselves. This means that while the linear memory can be indexed only using 32-bit integers, the data stored in the linear memory can indeed be 64-bit integers or 64-bit floating-point numbers. This also applies to local variables and values on the operand stack, which can be of any of the four supported types. In terms of the operand stack, it is neither strictly 32-bit nor 64-bit. Instead, it is capable of holding values of any of the supported types. When a 64-bit value is pushed onto the operand stack, it takes up 64 bits of space. When a 32-bit value is pushed, it occupies 32 bits. This flexibility allows for efficient execution of operations on different types of data. The distinction between the 32-bit index and the ability to handle 64-bit data types is crucial.

32-Bit Indexing WebAssembly's initial decision to use 32-bit indexes [20] was a strategic choice centered around the principle of portability across a broad range of hardware platforms. Additionally, it enables efficient memory usage, as 32-bit indexes consume less memory space compared to their 64-bit counterparts. However, the 32-bit addressing limit caps the accessible linear memory at 4 GiB, which may not suffice for more memory-intensive applications. Recognizing this limitation, there are ongoing efforts in the WebAssembly community to explore the introduction of 64-bit indexing, an initiative often referred to as the *wasm64* [51] project. Adopting 64-bit indexes would significantly increase the maximum accessible memory, opening the doors for more complex and resource-intensive applications to run on WebAssembly. However, such a transition is not without challenges. Besides the increase in memory usage due to larger index sizes, supporting 64-bit indexes could potentially exclude older or low-end hardware platforms. Nevertheless, extending WebAssembly to support 64-bit indexing is crucial for our project. We will explain more about this in Chapter 4.

WebAssembly, in its original design, is a sandboxed runtime that puts strong restrictions on the interactions between the WebAssembly module and the host environment. By default, a standard WebAssembly module cannot access external resources such as the filesystem or network connections. This restriction enhances the security and portability of WebAssembly, but also limits the range of applications that can be developed. WebAssembly itself does not include a standard library. Instead, the functionality is often provided by the host environment or supplementary libraries.

Browser For usage in the browser, WebAssembly provides mechanisms for interaction between its modules and JavaScript [35]. JavaScript developers can explicitly expose

functions to a WebAssembly module, allowing the module to call these functions as needed. WebAssembly is designed with a strong emphasis on the safety and security required in a web context, including restrictions on memory access and limited interaction with the host environment. Browsers typically deliver the WebAssembly runtime implementation, prominent examples being Google Chrome's V8 Engine [19] and Mozilla's SpiderMonkey Engine [33]. This runtime has to manage the stricter sandboxing requirements of the web environment and provide interfaces to web APIs. While the exact details can vary across different browsers, they all adhere to the same WebAssembly specification to ensure the portability of WebAssembly modules.

WASI On the other hand, the *WebAssembly System Interface* [14] (WASI) caters to non-browser applications. It was originally designed by the creators of the Wasmtime project as a modular system interface for WebAssembly, aiming to be portable across different operating systems. WASI provides APIs resembling POSIX, simplifying the process of porting applications written for Unix-like systems to WebAssembly. This means that WebAssembly code can run in a constrained environment without having access to sensitive system resources, adding an extra layer of security. These APIs allow controlled access to system resources such as filesystems and network connections, making WASI suitable for server-side or command-line applications. While WASI provides a comprehensive set of system calls that WebAssembly applications can use to interact with the host system, it does not automatically provide a convenient way for source languages to access these system calls. This is where *WASI-libc* [47] comes in as a C standard library implementation for WASI. It provides a mapping between the standard C library (libc) functions that many languages and programs rely on and the lower-level WASI system interface calls. WASI-libc is particularly useful when porting existing C or C++ programs to WebAssembly, which will be one of the main use-cases presented in this thesis. Crucially, WASI-libc only provides the definitions of the APIs, it does not include any specific implementations. This is why runtime WebAssembly implementations, such as Wasmtime, are a crucial part of the WASI ecosystem. They have the responsibility of providing implementations for the system interfaces specified by WASI. This implementation can vary across different host platforms, but the interface seen by the WebAssembly code remains the same.

2.2.4 Relationship with JavaScript in the Browser

The integration of WebAssembly within a web context is primarily managed by JavaScript. Through the WebAssembly JavaScript API [35], JavaScript controls the lifecycle of WebAssembly modules, from fetching and compiling the binary data to instantiation and function invocation. While the actual execution occurs within the

browser's WebAssembly engine, JavaScript serves as the orchestrator. This setup dictates that the design of WebAssembly modules must align with JavaScript's runtime characteristics, thus limiting the autonomy of WebAssembly modules.

Interoperability between WebAssembly and JavaScript requires transitions between their distinct execution contexts, leading to potential performance bottlenecks. This is because JavaScript and WebAssembly operate on different memory models, with JavaScript having a garbage-collected memory model and WebAssembly using a linear memory model. Such a difference can introduce complexities and inefficiencies when sharing data structures between the two.

In the context of our project, we focus primarily on enhancing the capabilities of WebAssembly and its interaction with the host environment through the WASI, without considering the potential effects on JavaScript interoperability. It is important to note that our modifications to the WebAssembly memory model could potentially make it more challenging to share memory with JavaScript.

2.2.5 Wasmtime: A Standalone WebAssembly Runtime

Wasmtime [12] is a standalone runtime for WebAssembly, built by Bytecode Alliance, an open-source community that includes Mozilla, Fastly, Intel, and Microsoft. Designed with performance, security, and configurability in mind, Wasmtime aims to provide a reliable execution environment for WebAssembly outside the traditional web context.

$$\text{WebAssembly Code} \xrightarrow{\text{Wasmtime}} \text{Cranelift IR} \xrightarrow{\text{Cranelift}} \text{Machine Code} \xrightarrow{\text{Execution}} \text{Result} \tag{2.1}$$

Figure 2.1: Wasmtime pipeline

The pipeline, visualized in Figure 2.1, is initiated by receiving the WebAssembly code. The Wasm code is then translated into *Cranelift IR*, an intermediate representation used by the Cranelift code generator. Cranelift is a code generator aiming to be fast, simple, and produce sufficiently efficient results. It is an integral component of the Wasmtime ecosystem and used as a backend for WebAssembly runtimes. While it performs some basic optimizations on the Cranelift IR, it generally expects the WebAssembly code it receives to be optimized already, either by the source language compiler (like LLVM for C/C++ or Rust), or by the developer if the WebAssembly code is hand-written. Subsequently, the Cranelift compiler translates the Cranelift IR into machine code either Ahead-Of-Time (AOT) or at runtime using Just-In-Time (JIT) compilation. This strategy allows Wasmtime to provide a good balance of startup time and runtime performance,

suitable for various use-cases and requirements. The generated machine code is then executed, yielding the program's result.

Wasmtime not only supports the standard WebAssembly feature set but also integrates with the WebAssembly System Interface (WASI) for interacting with the host environment. Furthermore, the development team is actively engaged in the WebAssembly standards process, and the runtime incorporates new proposals and features as they emerge. This ongoing commitment to staying current with the WebAssembly standards and its open, adaptable architecture make Wasmtime an ideal choice for projects that wish to experiment with extending or modifying the WebAssembly ecosystem, such as ours. Wasmtime also emphasizes security in its development. Built on top of the safety guarantees provided by Rust's language features, the runtime undergoes extensive review, ensuring that it is as robust as possible. Configurability is another significant aspect of Wasmtime. It provides default settings for common use-cases while allowing fine-grained control over resource consumption, such as CPU and memory, to suit diverse environments ranging from small devices to massive server deployments. A testament to its extensibility is Wasmtime's support for multiple backend architectures. Notably, it supports ARM64, which is of particular interest for our project.

2.3 LLVM

LLVM [27], which originally stood for Low-Level Virtual Machine, is an open-source compiler infrastructure project. One of its key advantages is its flexibility, as LLVM supports various programming languages and can generate optimized code for multiple target architectures. The central concept behind LLVM's language support is the use of a platform- and language-independent intermediate representation (IR). When a program written in a specific programming language is compiled using LLVM, it first goes through a compiler front-end. These front-ends are responsible for parsing and translating the syntax and semantics of a particular programming language into LLVM IR. The modular nature of LLVM allows for the development of language-specific front-ends. Examples of these frontends include clang[1], rustc[2], swiftc[3], and many others.

Once the program is in LLVM IR form, a series of modular analysis, transformation and optimization passes are applied. These passes can be used to enhance the program's performance, reduce its size, and enable other desirable transformations.

Arguably, function passes and module passes are most important for extending

[1] `https://clang.llvm.org`
[2] `https://www.rust-lang.org`
[3] `https://swift.org`

LLVM with new functionality. Function passes focus on optimizing and transforming individual functions within their respective scope. Module passes, on the other hand, operate on one translation unit at a time, optimizing the interactions between different functions and performing global optimizations. Importantly, these passes can also be used for semantic transformations that add new features to the generated code, not just optimizations. Our implementation is based on adding function and module passes that insert new instructions into the generated code.

Finally, LLVM's back-end code generators take the optimized LLVM IR and translate it into the target machine's native code. Crucially for our project, LLVM supports the WebAssembly architecture as one of its backend targets.

$$\text{C Code} \xrightarrow{\text{clang}} \text{LLVM IR} \xrightarrow{\text{LLVM.opt}} \text{LLVM IR'} \xrightarrow{\text{LLVM.codegen}} \text{Target} \tag{2.2}$$

Figure 2.2: LLVM pipeline

2.4 ARM64

In the realm of computer systems, a *computer architecture* [16] encompasses the design, organization, and operational structure of a computer system. It serves as the blueprint that describes how software and hardware technologies interact to create a computing environment. An architecture includes the instruction set, the processor's internal components, data types, memory address architecture, I/O mechanisms, and techniques for addressing memory. In this context, *data types* refer to the kinds of data that the computer can process, such as integers of various sizes, floating-point numbers, or complex types like vectors in Single Instruction, Multiple Data (SIMD) instructions. Furthermore, the architecture outlines the data paths, which are the physical and logical layouts where data is processed. These paths include circuits such as arithmetic logic units (ALUs), shifters, and registers, which perform operations on data, and the interconnections (buses) that transport data and control information between these components.

Historically, the x86 architecture, developed by Intel, has dominated the desktop and server markets [10]. However, the ARM architecture [5], standing for *Advanced RISC Machine*, has steadily been gaining popularity, especially in the realm of mobile and embedded systems. The ARM architecture was initially developed by ARM Holdings, a British company, in the 1980s [9]. One of the main advantages of the ARM architecture is its simplicity and energy efficiency. The architecture's *Reduced Instruction Set Computing (RISC)* approach, where it uses a smaller set of simple instructions, results in lower

power consumption compared to the *Complex Instruction Set Computing (CISC)* approach of x86 [45]. ARM's business model is unique in the sense that they do not manufacture the hardware chips themselves. Instead, they design the architecture and license it to other companies, who then manufacture the actual chips. This model allows ARM to focus solely on architecture design, thus promoting innovation and advancements in technology, without the burden of managing manufacturing supply chains or customer relations. Notably, not all features are included in every chip, as the manufacturers have the flexibility to choose which features to include based on their specific needs and target markets.

The ARM architecture has evolved over time, with major revisions labelled as ARMv#, such as ARMv7 and ARMv8. These revisions primarily focus on improving performance, enhancing security, and introducing so-called *hardware extensions*. These hardware extensions are new additions to the instruction set architecture, expanding the processor's functionality and providing innovative solutions to support a wider range of applications. Since ARM architectures come in both 32-bit (known as *AArch32*) and 64-bit (known as *AArch64*) versions, certain features may be exclusive to one version or the other. It is also important to note that the introduction of new hardware extensions does not always immediately result in improved performance or enhanced security. Instead, they provide new options and capabilities that compilers can target to optimize code specifically for these platforms. Therefore, the real-world benefits of these hardware extensions depend heavily on the compiler's ability to leverage these new instructions effectively.

One of the significant advancements in recent ARM architectures is the introduction of hardware extensions like *Pointer Authentication (PAC)* and *Memory Tagging Extension (MTE)*, which provide enhanced security and memory safety. PAC was introduced with ARMv8.3 and is already being implemented in several hardware devices, such as Apple's A12 and later mobile chips [4], utilized in many of the newer iPhone and iPad models, and Apple's M1 and later laptop chips [4], utilized in newer MacBooks. However, MTE was only introduced in ARMv8.5, and it is not yet widely available in hardware.

This creates the need for emulation tools like *QEMU* [38] for development and testing. QEMU, which stands for Quick Emulator, is an open-source machine emulator and virtualizer that allows software to run on a variety of hardware architectures irrespective of the host architecture. Although emulation incurs a performance overhead compared to running natively on hardware, it is still an acceptable approach for testing and development purposes, especially when hardware support for certain features is lacking. This is particularly relevant for our project, where we aim to demonstrate the potential of the MTE feature for memory safety before it becomes widely available in hardware.

Top Byte Ignore Feature ARM64 introduces an interesting feature that is crucial for enabling one of the hardware extensions we will look at further, MTE. In the 64-bit ARM architecture, although the virtual address space spans 64 bits, not all these bits are fully employed for memory addressing due to hardware-level specifications. An important distinction to remember is that these addressable bits correspond to Random Access Memory (RAM), not persistent storage. RAM is a type of memory used for temporary storage of data that is actively being processed by the CPU. Despite the fact that modern systems may have vast storage capacities running into multiple terabytes or even petabytes, the active data or working set that needs to be loaded into RAM and processed at any given time is often far smaller. Therefore, even with extensive storage capacities, systems generally only require a fraction of the 64-bit address space for direct memory addressing, rendering full utilization of this space unnecessary. This leaves room for the inclusion of additional features that save some kind of state within the upper bits of the address space. ARM64's design introduces a feature known as *Top Byte Ignore (TBI)* [8], which disregards the top byte of the address during the address translation process, leaving them available for other uses.

In the following sections, we will delve deeper into the specifics of Pointer Authentication and Memory Tagging Extension, since they are both utilized in our implementation.

2.4.1 Pointer Authentication Code (PAC)

In recent years, the rise of sophisticated cyber attacks has driven the need for advanced protection mechanisms in computer architectures. *Pointer Authentication* [39, 41] is one such innovative mechanism introduced by ARM in its AArch64 architecture. This feature is designed to authenticate pointers, thereby preventing attacks that rely on unauthorized modifications of pointer values.

The functionality of Pointer Authentication is rooted in its flexible utilization of the available unused bits within a pointer's representation. The exact number of bits dedicated to Pointer Authentication can vary, largely depending on other architectural features that may be in use. For example, when features like Memory Tagging Extension (MTE) are enabled, they share the pool of available unused bits, which affects the number of bits Pointer Authentication can use.

The Pointer Authentication extension uses the extra space in the pointer to embed a cryptographically generated signature, also known as a *Pointer Authentication Code (PAC)*, within the pointer itself. This signature is created using a *PAC key*, a portion of the pointer, and an additional value that depends on the context (such as the stack pointer). By housing this authentication code within the pointer, Pointer Authentication provides a means to validate the pointer's authenticity and maintain its integrity. It employs QARMA [39], a lightweight tweakable block cipher that provides cryptographic strength

even when truncated for very short signatures.

Key Management The PAC keys are essential for ensuring the robustness of the Pointer Authentication feature. They are dynamically generated by the hardware in an unpredictable manner, often using a hardware random number generator, and designed to be kept secret to mitigate potential risks of a security breach. If an attacker obtains these keys, they could potentially generate valid PACs and alter pointers in unauthorized ways. Therefore, the ARM64 architecture utilizes its concept of *exception levels (ELs)* [6] as a means of regulating access to these keys. These exception levels represent different tiers of privilege or trust within the system. EL0 is the lowest level, typically where user applications run, while higher levels (EL1, EL2, and EL3) are reserved for more trusted system software, such as the operating system kernel or hypervisor. The PAC keys are stored in system registers that are inaccessible from the lowest level, EL0. This means that compromised or malicious user-level applications cannot directly read or modify these keys. This ensures that they are accessible only to the most trusted parts of the system, minimizing the risk of their discovery by an attacker. To further enhance security, a total of five different keys are used, including distinct keys for instruction pointers and data pointers, as well as a general-purpose key. This approach of using multiple keys ensures that different types of pointers are independently protected, limiting the potential damage if one key were to be compromised. [39]

Instructions Concretely, ARM has extended the AArch64 Instruction Set Architecture (ISA) with the following new instructions [7]:

- The `PACIA` and `PACDA` instructions are used to compute and add a cryptographic signature to an instruction pointer and a data pointer, respectively. This signature is stored in the most significant bits of the pointer, which transforms the pointer into an illegal address until it is authenticated. These instructions also utilize a modifier, which is an additional input to the signature generation process that can provide contextual separation between different uses of the signed pointer. Notably, our project also employs the `PACDZA` instruction, a variant of `PACDA` that does not take a modifier as a parameter, instead consistently using the value zero. This makes `PACDZA` more generalized, offering simplicity in generating PAC signatures. A similar instruction, `PACIZA`, exists for the `PACIA` operation.
- The `AUTIA` and `AUTDA` instructions authenticate an instruction pointer and a data pointer, respectively. If the authentication fails, these instructions replace the signature with a specific pattern that keeps the pointer in an illegal state. However,

no error is immediately thrown. Instead, the error handling is decoupled from these instructions to remove the need for additional instructions for error handling. We should note the existence of `AUTDZA` and `AUTIZA`, counterparts to `AUTDA` and `AUTIA` respectively, which use the value zero as the modifier, similar to the function of `PACDZA` and `PACIZA`.

- The `XPACI` and `XPACD` instructions are used to strip the signature from a signed instruction pointer and a signed data pointer, respectively. These instructions essentially restore the pointer to its original, legal state.

Importantly, the actual error detection doesn't occur until an invalid pointer is dereferenced, at which point an illegal access exception is thrown. In other words, an exception arises when a pointer that has been either signed but not authenticated, or authenticated but not signed, is attempted to be dereferenced. This mechanism ensures the pointer remains unchanged throughout its lifecycle. The new instructions provide a flexible framework for developers and compiler designers to utilize Pointer Authentication effectively in the development of secure and efficient software. Notably, some of these instructions are encoded as architectural hint instructions and behave as *NOPs* (No Operation) to ensure binary backwards compatibility, allowing older ARM processors to execute binaries compiled with the new instructions.

Preventing attacks The layout of memory on a stack, where data and return addresses are stored linearly and in close proximity, provides an opportunity for malicious exploits. In attacks such as *Return-Oriented Programming (ROP)* and *Jump-Oriented Programming (JOP)*, malicious actors exploit the stack's structure, overflowing buffers to overwrite return addresses, thereby hijacking control flow to execute chosen code fragments. To counter this, software stack protection mechanisms were developed [39]. These involve the compiler placing a randomized *canary* value between the return address and the stack buffers upon function entry. This canary value is then checked for any modifications before the function returns. However, while this method can mitigate certain types of buffer overflow attacks, it remains vulnerable to bypassing tactics if there are memory disclosure vulnerabilities and presents some performance overhead. By using Pointer Authentication, compilers can sign and authenticate these pointers efficiently to verify their integrity before use. If a pointer has been tampered with, its cryptographic signature will not match the expected value, alerting the system to the malicious activity. This significantly increases the difficulty of tampering with pointers without detection, effectively raising the bar for successful stack-based attacks.

2.4.2 Memory Tagging Extension (MTE)

The *Memory Tagging Extension (MTE)* [8] is another advanced feature incorporated into the ARM AArch64 architecture, designed to safeguard against both malicious attacks and accidental programming bugs related to memory safety. At a high-level of abstraction, MTE provides a "lock and key" system from memory access. In this analogy, locks are set on memory regions, and keys are provided by the memory addresses used to access these regions. If the key matches the lock, the memory access is permitted, otherwise an error is thrown. This lock and key mechanism is implemented in MTE through the use of *tags*.

MTE makes use of the unused bits in a pointer's representation thanks to the Top Byte Ignore (TBI) feature. However, rather than signing pointers like PAC, MTE assigns a 4-bit *tag* to both the memory addresses (or pointers) and the memory regions they point to. This can be imagined as equivalent to assigning specific keys to the memory addresses and specific locks to the memory regions. These tags are stored in bits 56 to 59 of the 64-bit memory address. Meanwhile, the tags for the memory regions are stored separately in a dedicated memory region managed by the hardware. When a memory region is accessed through a pointer, the hardware checks that the tags of the pointer and the memory region match. If there is a mismatch, the system throws an MTE exception, signaling a potentially illegal memory access.

Importantly, memory regions are tagged by adding four bits of metadata to each 16 bytes of physical memory. This 16-byte granularity is know as the *Tag Granule*. The choice of this granularity represents a deliberate design balance between precision and performance. With a 16-byte granularity, the system can protect smaller regions of data, thereby enhancing security by preventing more sophisticated forms of memory corruption, such as overflows. On the other hand, larger granularities can potentially have less performance overhead, but do so at the expense of precision in memory protections.

Instructions ARM has extended the AArch64 Instruction Set Architecture (ISA) with new instructions [7] to facilitate the use of MTE. Here, we highlight those relevant to our implementation:

- The `IRG` (Insert Random Tag) instruction generates a new random tag, chosen randomly from the 16 possible values, and inserts it into a specified pointer. This instruction is used when we want to assign a new, unique tag to a memory address.
- The `STG` (Store Allocation Tag) instruction writes the tag from a pointer to the corresponding memory region. This instruction is used to tag a memory region

with a specific tag so that it can only be accessed by pointers carrying the same tag.

- The `ST2G` (Store Allocation Tags) instruction is similar to the `STG` instruction but operates on two consecutive memory regions. This is used when we want to assign the same tag to two adjacent memory regions.

These three instructions together form the basis for tag management in MTE, allowing us to assign tags to both pointers and memory regions.

Synchronous and Asychronous Modes Another compelling aspect of MTE is its support for both *synchronous* and *asynchronous* operation modes [2]. The synchronous mode, while slower in execution, provides more accurate diagnostics. This precise detection stems from the fact that any memory tagging faults are instantly flagged at the precise location of occurrence, triggering an immediate OS signal. On the other hand, the asynchronous mode trades off some diagnostic precision for faster runtime speeds. In this mode, faults due to invalid memory accesses are not flagged instantaneously. Instead, they are typically recognized only at the next transition from userspace to the kernel, a context switch. This delayed fault recognition leads to less accurate diagnostic information. The received backtrace corresponds not to the moment of the actual invalid memory access, but rather to the subsequent context switch when the fault was finally detected.

Preventing attacks As discussed in Section 2.1, both spatial and temporal safety violations pose significant challenges to ensuring memory safety. MTE provides us with the necessary tools to prevent these kinds of memory safety errors in our compiler implementations, which we will discuss in more detail in Chapter 5.

Tradeoffs However, MTE is not without its trade-offs. For one, its required 16-byte granularity means that memory allocations need to be padded up to a multiple of 16 bytes, which may lead to increased memory usage overhead. To minimize this overhead, there are certain best practices [8] developers can follow:

- Avoid over-allocating address space that never has data written to it. MTE may still need to assign tags to such regions, which could lead to unnecessary overhead.
- Avoid excessive de-allocation and re-allocation. Since using MTE has a fixed cost for memory de-allocation and re-allocation, frequent changes can lead to increased overhead.

- Avoid large fixed-size allocations on the stack. Reducing such allocations can help minimize the overhead of protecting the stack.

2.4.3 Using PAC and MTE Together

The dynamic bit allocation strategy in ARM64, allowing features like PAC and MTE to use available unused bits, presents an interesting challenge in balancing security and functionality. While each hardware extension is individually effective against specific types of attacks, they all depend on the limited extra space in the address representation. Their simultaneous use can potentially diminish each feature's key space, reducing the degree of uniqueness in their respective security identifiers. This could, theoretically, decrease the usefulness of the hardware extensions in defending against successful brute-force or cryptographic attacks. Collectively, hardware designers, operating system developers, and security researchers carry the significant task of determining the optimal configuration of these hardware extensions. One can argue that, even if the key space for each feature might be reduced due to sharing, the layered approach of multiple security measures could still ensure robust system security. The interplay between these features underlines the complexities of modern system security and the necessity for a tailored approach, which adjusts to specific use-cases and threat models.

3 Overview

Memory safety is an essential property of robust, secure computing systems. However, to enforce memory safety in a system, we often incur a performance cost that can impede efficiency and affect the overall user experience. WebAssembly, in this context, offers an interesting domain to explore due to its properties of fast execution and safe, portable code. However, ensuring memory safety in such an environment, without significantly compromising performance, remains a challenging endeavor. The challenge lies in striking a balance between preserving memory safety and maintaining the high performance inherent to WebAssembly.

Thus, the central question of this research paper arises:

> How can we leverage ARM's hardware extensions to provide memory safety in WebAssembly without incurring significant performance costs?

To provide an answer, we intend to leverage recent ARM hardware extensions, specifically Pointer Authentication (PAC) and the Memory Tagging Extension (MTE). These features introduce a new spectrum of security capabilities at the hardware level, providing an opportunity to maintain system performance while enhancing safety when running WebAssembly programs on the AArch64 architecture.

In the upcoming sections, we will further elaborate on our strategy. We will begin by setting the stage with our design goals in Section 3.1, followed by a detailed discussion on the design in Chapter 4, as well as a closer look at the actual implementation in Chapter 5. The paper will continue with an evaluation of our solution, in which we measure the performance impact and assess how well it meets our stated goals. Lastly, we will present a survey of related work, positioning our contribution within the larger context of efforts to enhance memory safety and performance in WebAssembly.

3.1 Design Goals

Safety The primary objective of our project centers around enhancing the *safety* of WebAssembly. Therefore, our aim is to catch as many memory safety bugs as possible at runtime. We want to ensure that WebAssembly programs running on our software stack are safeguarded against common programming errors that can lead to unexpected behavior or severe security vulnerabilities.

Performance While safety is our top priority, we also value *performance* highly. We want our solution to not only be viable for testing purposes but also for deployment in production systems. Thus, high performance with minimal overhead is a crucial benchmark for us.

Usability *Usability* is another key area we want to focus on, specifically for developers who are looking to compile their source code into WebAssembly and execute it using our toolchain. In this context, usability refers to offering a seamless and straightforward experience for developers. This goal is twofold. Firstly, we strive to ensure that developers do not need to modify their input source code at all to work with our software stack. Secondly, we aim to support as many programming languages as possible as input with our system.

3.2 System Components

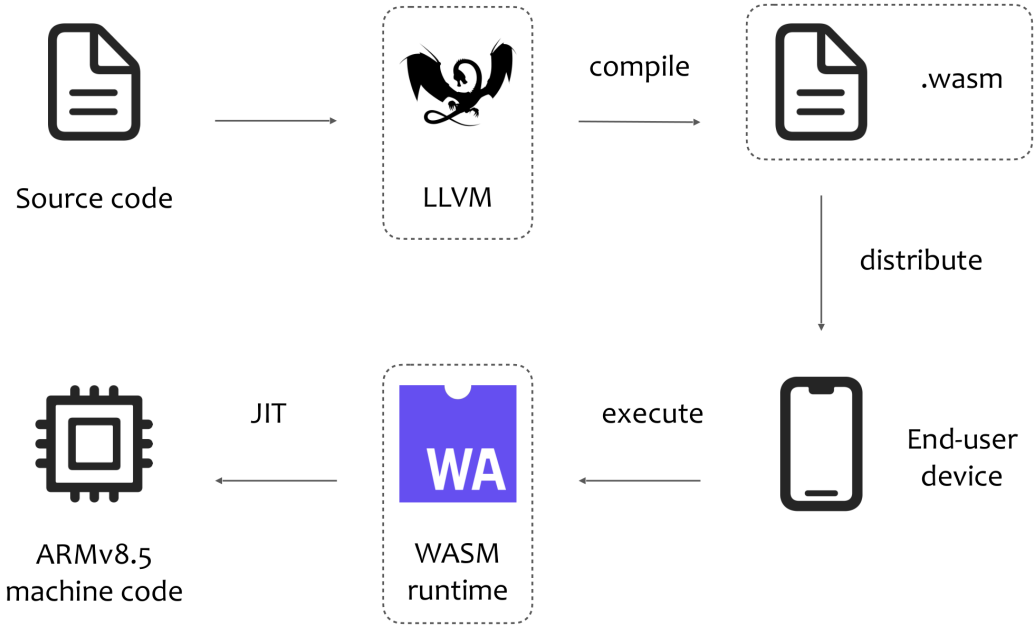


Figure 3.1: System components

LLVM The LLVM compiler toolchain forms a key component of our system. Its primary role is to take the input source code and compile it into WebAssembly. While we utilize Clang as our LLVM frontend in our current prototype, in the future it could be replaced with any suitable LLVM frontend as needed. It is important to note that in our system, we also leverage LLVM to analyze the input source code at the LLVM IR level to determine where ARM64 PAC and MTE instructions should be inserted. The specifics of this will be further discussed in Section 4.

Distribution Thanks to WebAssembly's universal binary format, we can avoid recompilation for each end-user device's architecture. Instead, we can compile the source code into WebAssembly once using LLVM, and distribute the same binary across a multitude of devices, each potentially having a different architecture. The execution is then facilitated by a WebAssembly runtime on the user's device.

Wasmtime Wasmtime, our chosen WebAssembly runtime, is employed to execute the compiled WebAssembly code on the user's device. This open-source runtime enables seamless execution of our WebAssembly code across numerous supported systems. However, our incorporated safety features only become active when executed on an ARM64 device that supports both PAC or MTE.

Execution Environment The ideal execution environment for our system would be an ARMv8.5 hardware setup supporting both the Memory Tagging Extension (MTE) and Pointer Authentication (PAC). However, as such hardware currently does not exist (particularly the MTE supporting hardware), we resort to QEMU for emulation. This enables us to test and validate the functionality of our added features.

4 Design

In this chapter, we delve into how we aim to fulfill the design goals in Section 4.1. We will also explore the details of our extensions to WebAssembly, LLVM, and Wasmtime in Section 4.2. It is important to mention that our current prototype is primarily designed to accommodate the C programming language as the input source language. Therefore, there might be certain design and implementations details outlined in this section and in Section 5 that are specific to C.

4.1 Approach to Fulfilling the Design Goals

In this section, we outline the measures taken to meet our design goals as described in Section 3.1.

Safety and Performance To achieve both safety and performance, we adopt a hardware-based approach rather than a software-based one. By leveraging the recent hardware extensions offered by ARM, we aim to significantly reduce runtime performance overhead while catching as many memory safety bugs as possible. These hardware-based approaches promise to incur less runtime performance overhead compared to their software-based counterparts.

Usability Our approach to ensuring usability involves minimal intrusion into the developers' code and offering support for as many input languages as possible. To make our software stack user-friendly, we focus our modifications on the compiler and runtime, specifically LLVM and Wasmtime, rather than the source code itself. By doing so, developers can seamlessly integrate our software stack into their development process without the need to modify their source code. All that is required from their end is to recompile their programs using our modified compilers and execute the obtained WebAssembly in our modified runtime. To support as many programming languages as possible, we chose LLVM as our compilation toolchain. LLVM is known for its widespread extensibility and compatibility with numerous programming languages, which provide LLVM frontends. This decision allows our software stack to accommodate a broad range of input languages, thereby increasing its usability. Furthermore,

LLVM supports WebAssembly as a target, a crucial requirement for our project.

4.2 Extensions

This section outlines the extensions made to WebAssembly, LLVM, and Wasmtime.

4.2.1 WebAssembly

First of all, it is crucial to note that our project works solely with *wasm64*, not the original *wasm32*. As we have already hinted to in previous chapters, this is due to the nature of ARM64 instructions which store additional metadata in the upper bits of a 64-bit pointer. To preserve this metadata, we utilize 64-bit indexes, i.e. wasm64.

MTE For the part of our design that is centred around the Memory Tagging Extension (MTE), we introduce two new concepts: *segments* and *tagged indexes*. Segments correspond to memory regions that are protected against spatial and temporal memory violations, by being tagged with an MTE tag. These segments can represent any memory allocations, be it on the C stack or heap. The tagged index, on the other hand, can be imagined as the pointer or virtual memory address that points to the segment, which is also tagged with an MTE tag. However, since WebAssembly uses indexes instead of pointers, the tagged index is the relative index from the start of the linear memory pointing to the segment.

PAC Our design strategy for implementing pointer authentication involves signing pointers before storing them in a memory location and authenticating them after loading them from a memory location. This mechanism mitigates one type of spatial memory violation - where a memory location storing a pointer gets maliciously overwritten, potentially via a buffer overflow. When the pointer is subsequently loaded from this compromised memory location, we can no longer guarantee that it is our original pointer. By signing the pointer before storing it, we ensure that authenticating the loaded pointer will fail if the memory location, it was loaded from, was overwritten.

New Instructions Alongside these new concepts, we also introduce new instructions to WebAssembly:

- `segment.new(index: i64, size: i64) -> i64`
  This instruction takes the index pointing to a segment, as well as the size of the segment, as parameters. The purpose of this instruction is to protect the memory

segment by ensuring access is only permitted via the associated index that is returned. Internally, this is achieved by assigning a random tag to the index, and consequently tagging the segment with the same random tag. This is essential in allowing us to recognize spatial memory violations, because any access to the segment that does not go through the tagged index should be recognized and denied. However, the protection offered remains probabilistic due to the randomness of tag generation.

- `segment.free(index: i64, size: i64)`
  This instruction is meant to be called once the segment is no longer used and should be freed. This is essential for identifying temporal memory violations, such as *use-after-free* bugs, which we can achieve by tagging the segment with a special MTE tag. The specifics about this special tag will be elaborated upon in Section 4.2.3 concerning the modifications made to Wasmtime. In essence, the goal is for the index and memory region to have differing tags, enabling the detection of use-after-free bugs. Importantly, this instruction does not alter the index, only the segment.

- `i64.pointer_sign(index: i64) -> i64`
  This instruction signs an index with a cryptographic signature, and returns the signed index.

- `i64.pointer_auth(index: i64) -> i64`
  This instruction authenticates an index according to its cryptographic signature, and returns the authenticated index.

In Wasmtime, these extensions to the WebAssembly standard must be enabled using a feature flag.

4.2.2 LLVM

In order to achieve memory safety in WebAssembly, we extended the LLVM WebAssembly target and introduced LLVM passes that operate specifically on the LLVM Intermediate Representation (IR) if the target is WebAssembly.

MTE The primary objective of our memory safety analysis revolves around identifying the creation, resizing, and deletion of segments. To do so, we examine all allocation functions corresponding to C's heap-based allocations [28], which are `malloc`, `calloc`, `realloc` and `free`, as well as stack-based allocations. For each of these allocation-related function calls, we have created a custom wrapper function that extends them

with our additional memory safety features. Therefore, once we have found each of the instances of these allocation function calls, we replace them directly with our corresponding custom wrappers. For the allocation functions (malloc, calloc, realloc), our wrappers insert the `segment.new` instruction after the wrapped allocation. As for the `free` wrapper, responsible for de-allocation, it first inserts the `segment.free` instruction, and then calls the regular C `free`. There are several reason why we need these wrappers:

- The Memory Tagging Extension (MTE) requires tagging memory locations at a 16-byte granularity. This implies that each segment must be 16-byte aligned to be eligible for tagging. To accommodate this, each allocation's size must adhere to the 16-byte alignment. We have implemented this within our wrapper by calculating the aligned size, which is the smallest multiple of the alignment (a minimum of 16, but could be more if requested by the user, e.g., with `aligned_alloc`) that is not less than the requested size. This aligned size then becomes the real amount of memory we allocate.
- Inserting the `segment.free` instruction necessitates knowing the size of the memory region we aim to free since the instruction uses this as a parameter. To facilitate this, we maintain a data structure that retains the metadata for each memory location, which includes the pointer to, and the size of, the memory location. This data structure is updated with each new allocation (new elements are inserted) and on *free* calls (elements are deleted).

PAC Implementing pointer authentication as envisaged involves identifying locations in the input program where a pointer is being stored to or loaded from a memory location. Once these locations are found, we can simply insert a `pointer_sign` instruction before the store instruction, and a `pointer_auth` instruction following the load instruction.

During our research, we discovered certain requirements that would be necessary for a successful pointer authentication implementation.

Requirement 1 *Pointers that are signed must be authenticated before being dereferenced.*

Requirement 2 *Pointers that are never signed are strictly prohibited from being authenticated before being dereferenced.*

Violation of these requirements results in unexpected PAC exceptions at runtime, even if the compiled program does not have any apparent vulnerabilities that PAC is designed to guard against. This is due to the inherent characteristic of PAC: Once a

pointer is signed, the PAC signature bits are corrupted, causing subsequent dereferences of the pointer to trigger a PAC exception. The corrupted pointer can only be restored via PAC authentication, which returns the upper bits to their valid state. Conversely, authentication will invariably fail for pointers that have not been signed, as they lack a PAC signature. Simply put, we must ensure that every pointer signing is matched with an authentication, and if we cannot guarantee both operations, we cannot implement either.

This initial requirement has implications for our LLVM analysis, prompting us to establish a subsequent requirement.

Requirement 3 *We are strictly prohibited from performing pointer authentication on memory locations either received from or provided to external dependencies of the program being compiled. These external dependencies include function declarations that are defined only at the link or runtime stage, and thus do not form part of the compiled program.*

These external dependencies potentially violate the Requirements 1 and 2 by interacting with memory locations containing pointers. If we sign a pointer, store it in the memory location, and then pass that memory location to an external dependency, our LLVM analysis cannot predict the actions of the external dependency. If the external dependency dereferences the pointer without authentication, this violates Requirement 1, because we signed a pointer but never authenticated it. We cannot insert an authentication instruction into the external dependencies' codebase, as we lack access to it at the time of compiling our program. Similarly, if we receive a memory location containing a pointer, inserting an authentication could potentially violate Requirement 2, since we cannot determine whether the external dependency signed the pointer.

Earlier in Section 2.2.3, we discussed our intentions to leverage WASI-libc to compile C code to WebAssembly, thereby providing the necessary low-level system interfaces. However, the use of WASI is at odds with Requirement 3, since WASI represents an external dependency that interacts with memory locations storing pointers. This is because the WASI-libc provides an interface of system-level function declarations, but the definitions of which are provided only at runtime by the WebAssembly implementation.

To resolve this issue and adhere to Requirement 3, our LLVM analysis has to exclude memory locations associated with external functions from pointer authentication. Identifying these *external functions* is thus a crucial aspect of our analysis. Therefore, we provide a precise definition of what constitutes an external function relative to a given entity.

Definition 2 *An external function, relative to the entity* E*, is a function that is declared but not defined within* E*.*

The distinction between entities is crucial as LLVM IR mostly operates at three levels of abstraction: function, module (which can be imagined as a self-contained unit of compilation representing a single source code file), and the entire program (which contains all modules). In theory, it might be feasible to perform an analysis on the entire program after all the modules have been merged, a process known as a *Link-Time Optimization* pass in LLVM. While this approach would be ideal in terms of safety, allowing us to sign and authenticate as many memory locations storing pointers as possible, we currently have only function and module passes in our prototype, as these are the most commonly used passes in LLVM and better suited to early experimental implementation.

- **Function-level granularity**:
  At this level, all functions are considered external, as no function can be defined within another function in LLVM. Consequently, we do not perform pointer authentication on a memory location if the location is passed to another function, or if the location is the return value of a function. We must also account for aliases of the memory location, which require analysis equivalent to that performed on the original memory location. While this approach provides some safety measures, its effectiveness is limited in scenarios where memory locations are passed between functions.

- **Module-level granularity**:
  In this design, a function is considered external if it is declared but not defined in the current module. Consequently, we avoid performing pointer authentication on any memory location passed to or returned from an external function. This rule also applies to *transitive* interactions - cases where a memory location is passed to a non-external function, which in turn passes the memory location to an external function. To capture these indirect interactions, a recursive search algorithm is employed, which we elaborate on in Section 5.2.1. Despite requiring a more in-depth LLVM analysis, this strategy allows for the signing and authentication of a greater number of memory locations, fulfilling our objective of pointer authentication within the defined requirements.

4.2.3 Wasmtime

As we have already mentioned in the section on the LLVM extensions, for the memory safety guarantees of our project to be activated, we must enable the 64-bit WebAssembly mode in Wasmtime using the `-wasm-features=memory64` flag.

While we have leveraged LLVM to emit our custom WebAssembly instructions, these alone are insufficient to ensure memory safety. Instead, these instructions require

implementation and support within the WebAssembly runtime. This step enables the runtime to compile the custom WebAssembly instructions to architecture-specific ones.

Cranelift IR *Cranelift* serves as the code generator within Wasmtime. Much like LLVM IR, *Cranelift IR* functions as an intermediate representation during the compilation process. There are two important translations in our Wasmtime toolchain: Wasmtime first translates the input WebAssembly code into Cranelift IR, which Cranelift then uses to generate efficient machine code. Consequently, we had to introduce new Cranelift IR instructions. The instructions we added directly correspond to the AArch64 instructions we are employing, namely `IRG`, `STG`, `ST2G`, `PACDZA`, and `AUTDZA`. Whenever these Cranelift IR instructions are generated, they are translated into AArch64-specific instructions, causing issues when targeting an architecture other than AArch64. However, we chose this design approach with the intention that users should enable our enhanced memory safety support in LLVM only if they plan to target an AArch64 machine from Wasmtime.

We will now delve into how our extended Cranelift translates the input WebAssembly code into Cranelift IR instructions. As we have established, these IR instructions mirror the final AArch64 machine code. Therefore, we will synonymously use these two types of instructions and refer to specific instructions by their AArch64 names, which we introduced in Section 2.4:

- `segment.new(index: i64, size: i64) -> i64`
  The `segment.new` operation tags the index with a newly generated tag via the `IRG` instruction. Subsequently, it tags the memory region pointed to by the index and with the given size with the same newly generated tag using a series of `STG` instructions. Further details on optimization strategies applied for tagging memory regions are discussed in Section 5.1.1. After tagging the memory region, the function pushes the tagged index, which we received from `IRG`, back onto the Wasm stack as it serves as the return value of the Wasm instruction.

- `segment.free(index: i64, size: i64)`
  When configuring the linear memory to work with MTE, we can provide Linux with a tag that should never be generated by the `IRG` instruction. Linux itself classifies `0b0000` as the default (free) tag, hence uninitialized memory is tagged with this tag by default by Linux. Therefore, we also chose the tag `0b0000` as the *free tag*, which is used as the indicator for freed memory. When the `segment.free` instruction is called, we tag the entire memory region to be freed with the special free tag. We also call this process *untagging* a memory region. The same strategy and optimization methods used for memory region tagging in `segment.new` are applied here as well.

- `i64.pointer_sign(index: i64) -> i64`
  In the `i64.pointer_sign` operation, the index is first popped from the Wasm stack. Then, the `PACDZA` instruction is executed with this index as its parameter. The output - a signed version of the index - is then pushed back onto the stack, effectively replacing the original index on the top of the stack with its signed variant.

- `i64.pointer_auth(index: i64) -> i64`
  The `i64.pointer_auth` instruction is handled similarly to the `i64.pointer_sign` instruction. It pops the index from the stack, executes the `AUTDZA` instruction with the index as the input, and finally pushes the authenticated index back onto the stack, replacing the original index with its authenticated variant.

Masking the MTE Bits With MTE enabled, the indexes will have their most significant bits set with the MTE tag. However, this could create issues when integer comparisons involving the index value are performed since the tag skews the index's integer value, making it appear significantly larger. Such integer comparisons are frequent within Wasmtime, especially when performing out-of-bounds checks for linear memory. These checks are designed to ensure that the index does not exceed the total size of the linear memory. Consequently, we must temporarily remove the MTE bits from the index when these checks are performed. Notably, this operation is unnecessary for the PAC signature. This is because the index is authenticated directly after it is loaded from a memory location. A successful authentication restores the index to a valid address. Hence, the index only ever contains the PAC signature when stored in memory and has it removed once it is used in the program.

Elimination of Runtime Out-Of-Bounds Checks In addition to protecting individual memory segments, MTE offers the potential to bypass the need for runtime out-of-bounds checks in Wasmtime. For the 64-bit variant of WebAssembly as handled in Wasmtime, each linear memory access currently undergoes a bounds check to verify the accessed index stays within the linear memory's limits, incurring a notable runtime overhead. To address this, we propose to tag the entire linear memory with a unique *linear memory free tag*. Consequently, we distinguish between two specialized free tags: the *linear memory free tag* and the *default free tag*. The latter remains unchanged, acting as the default tag for addresses and memory regions in Linux (with a default value of 0), and is used to tag the memory of the WebAssembly runtime itself. By employing different tags for the linear memory and regions outside of it, a clear separation is established, causing MTE traps on out-of-bounds violations. Accommodating this new strategy also requires that segments within the linear memory are tagged with

the linear memory free tag when freed. While the implementation of this feature is still underway, its potential suggests a promising path for enhancing the runtime performance of 64-bit WebAssembly on ARM64 devices equipped with MTE.

4.3 Showcasing Attack Protection with Practical Examples

Now we will provide real malicious C code examples that we are able to protect with our software stack.

4.3.1 MTE

Spatial Memory Violations

C Code We illustrate an *array-out-of-bounds access*, a particular type of spatial memory violation, through the following C code. This program creates an integer array of size 4 on the stack, and then attempts to access the non-existent fifth element in the buffer, resulting in an out-of-bounds access.

```c
#include <stdio.h>

int main() {
    int buffer[4];
    printf("%d", buffer[4]); // array-out-of-bounds access

    return 0;
}
```

Listing 4.1: C code example of an array-out-of-bounds bug

LLVM IR We compile this code using our modified LLVM with the Clang frontend, generating the following LLVM IR code. The code snippet presented here is a stripped-down version, with debugging metadata left out for brevity.

```
define i32 @__original_main() {
entry:
  %buffer = alloca [4 x i32], align 16
  %buf = call ptr @llvm.wasm.segment.new(ptr %buffer, i64 16)
  %arrayidx = getelementptr inbounds [4 x i32], ptr %buf, i64 0, i64 4
  %val = load i32, ptr %arrayidx, align 16 ; array-out-of-bounds access
  %call1 = call i32 @printf(ptr @.str, i32 %val)
```

```
  call void @llvm.wasm.segment.free(ptr %buf, ptr %buffer, i64 16)
  ret i32 0
}
```

Listing 4.2: LLVM IR code example of an array-out-of-bounds bug

The illustrated LLVM IR code begins by allocating a stack buffer for four integers. This buffer is then protected via our custom WebAssembly instruction, `@llvm.wasm.segment.new`. The `getelementptr` instruction that follows calculates a pointer to a non-existent fifth element in the buffer, stepping outside its boundaries. The actual out-of-bounds access takes place when a value is loaded from this location, embodying the array out-of-bounds bug in our program. Finally, the custom WebAssembly instruction `@llvm.wasm.segment.free` is used to free the buffer.

WebAssembly Code The primary output from LLVM is the WebAssembly code. It is worth noting that even after LLVM optimizations, the complete generated WebAssembly textual format file comprises thousands of lines, as it includes a significant portion of the libc required to compile printf and the data structures our allocation wrappers use to store metadata. Therefore, we focus only on the most crucial parts of the WebAssembly code.

```
(func $__original_main (result i32)
  ...
  local.get 0
  i64.const 16
  segment.new align=1
  local.set 1
  ...
  local.get 1
  i32.load offset=16 ; array-out-of-bounds access
  ...
  local.get 1
  i64.const 16
  segment.free align=1
  ...
)
```

Listing 4.3: WebAssembly code example of an array-out-of-bounds bug

Listing 4.3 shows the `__original_main` function: The function initially pushes the index of the stack-allocated array and the size of the memory segment onto the stack. These parameters are then passed to our custom `segment.new` instruction, creating a

protected segment for the specified memory region. The index of this newly created memory segment is then stored in a local variable. Later on, the function attempts to load a value from an offset beyond the array's boundary, specifically at the 16-byte mark, which is beyond the size of the four-integer array. This offset results in an array-out-of-bounds access, highlighting the array-out-of-bounds bug we are investigating. Finally, our custom instruction `segment.free` is executed, taking the index of the formerly protected memory segment and its size as parameters.

Execution in Wasmtime Upon running this WebAssembly code using Wasmtime, we trigger a trap, as shown below.

```
Error: failed to run main module ‘demo.cwasm‘

Caused by:
    0: failed to invoke command default
    1: error while executing at wasm backtrace:
           0: 0x270 - main
                          at demo.c:5:18
           1: 0x1eb - <unknown>!_start
    2: memory fault at address 0x11320 in linear memory of size 0x20000
    3: wasm trap: got memory tagging extension (mte) fault
```

Listing 4.4: Wasmtime error message for program with array-out-of-bounds access

As expected, Wasmtime encounters a runtime exception and halts the execution, preventing the usual undefined behavior that follows an array-out-of-bounds access bug. This bug is detectable because the array is the only memory region allocated on the stack. Consequently, the region that follows the array remains uninitialized and, thus, untagged. This results in a tag mismatch between the tagged index and the untagged segment being accessed.

Due to our implementation (elaborated further in Section 5.1.2), Wasmtime not only recognizes the exception, but it also precisely identifies it as an MTE exception. Moreover, it even pinpoints the correct file and line of the fault location in the input C code, substantially aiding in error debugging.

Temporal Memory Violations

C Code To demonstrate how our design safeguards against temporal memory violations, we showcase a simple C program that includes a use-after-free bug in Listing 4.5. This program allocates memory for an integer, stores a value at this address, frees the memory, and crucially, attempts to access the memory location after it has been freed.

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *x = malloc(sizeof(int));
    *x = 42;
    free(x);
    printf("%d", *x); // use-after-free bug

    return 0;
}
```

Listing 4.5: C code example of a use-after-free bug

LLVM IR Using our customized LLVM with Clang, we compiled the original code into the concise LLVM IR snippet below, removing debugging metadata.

```
define i32 @__original_main() {
entry:
  %x = call ptr @__wasm_memsafety_malloc(i64 16, i64 4)
  call void @__wasm_memsafety_free(ptr %x)
  %x_val = load i32, ptr %x, align 4 ; use-after-free bug
  %call1 = call i32 @printf(ptr @.str, i32 %x_val)
  ret i32 0
}
```

Listing 4.6: LLVM IR code example of a use-after-free bug

This code shows the replacement of `malloc` and `free` calls with our custom wrappers `__wasm_memsafety_malloc` and `__wasm_memsafety_free`. `__wasm_memsafety_malloc` takes parameters for alignment, here defaulting to 16 due to the MTE tag granule required alignment, and the size of the memory to allocate. Note that even though we request 4 bytes (`sizeof(int)`), the actual allocated size will be 16 bytes due to the alignment requirement.

WebAssembly Code Now, we focus on the three key functions in the generated WebAssembly code.

```
(func $__wasm_memsafety_malloc (;7;) (type 8) (param i64 i64) (result i64)
    ...
    call $aligned_alloc
```

```
    ...
    segment.new align=1
    ...
)
```

Listing 4.7: WebAssembly code example of malloc wrapper

The function `__wasm_memsafety_malloc` from Listing 4.7 takes the desired memory alignment and size of memory to be allocated, while `aligned_alloc` performs the actual allocation. The `segment.new` instruction then signals to Wasmtime that the new segment needs to be tagged.

```
(func $__wasm_memsafety_free (;8;) (type 9) (param i64)
    ...
    segment.free align=1
    ...
    call $free
    ...
)
```

Listing 4.8: WebAssembly code exmample of free wrapper

Listing 4.8 shows the function `__wasm_memsafety_free`. This function takes a data index as a parameter, uses the `segment.free` instruction to untag the segment, and executes the standard C `free` function on the index.

```
(func $__original_main (;9;) (type 10) (result i32)
    (local i64 i64)
    ...
    i64.const 16
    i64.const 4
    call $__wasm_memsafety_malloc
    local.tee 1
    call $__wasm_memsafety_free
    local.get 0
    local.get 1
    i32.load ; use-after-free bug
    ...
)
```

Listing 4.9: WebAssembly code example of a use-after-free bug

The `__original_main` function in Listing 4.9 can be described as follows: After setting up the stack and declaring local variables, the function pushes the alignment

and size of the memory to be allocated onto the stack. These parameters are then passed to `__wasm_memsafety_malloc`. The returned memory address is stored in a local variable and freed using `__wasm_memsafety_free`. The function then loads the value from the freed memory location, which constitutes the use-after-free bug. Finally, the function concludes with several operations to update the stack pointer and return from the function.

Execution in Wasmtime On executing this WebAssembly code with Wasmtime, we once again encounter an MTE trap. Helpfully, it pinpoints the fault precisely to `demo.c:8:18`.

4.3.2 PAC

We already explained how PAC is employed to prevent certain buffer overflow attacks, a vulnerability that the Memory Tagging Extension (MTE) is also designed to eliminate. However, this does not make our pointer authentication protection irrelevant. As previously highlighted, MTE is currently not available in real hardware, whereas PAC is. Thus, devices that support PAC, but not MTE, can still achieve improved memory safety, albeit with weaker guarantees.

Spatial Memory Violations

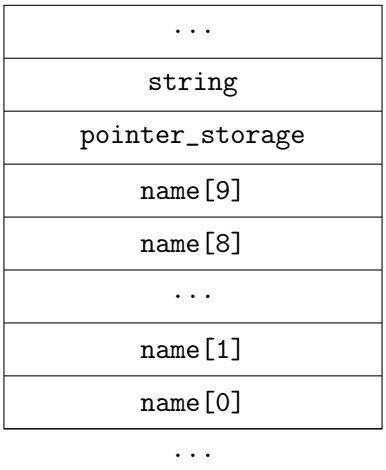
C Code To illustrate how our usage of Pointer Authentication protects against a specific kind of spatial memory violation, we showcase a simple C program that includes a potential buffer overflow in Listing 4.10. Note that the MTE protected segments are disabled here to be able to showcase the attack.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    char *string = "Hello␣World!";
    char **pointer_storage = &string;
    char name[10];

    printf("What␣is␣your␣name?\n");
    scanf("%s", name); // potential buffer overflow
    printf("Hello␣user␣%s!\n", name);

    char *loaded_string = *pointer_storage; // failed authentication
```

```
    printf("String␣protected␣with␣PAC:␣%s\n", loaded_string);

    return 0;
}
```

Listing 4.10: C code example of PAC protection

Bottom of the stack (higher memory addresses)



Top of the stack (lower memory addresses)

Figure 4.1: Stack layout and potential buffer overflow in PAC-protected program

In this program, a string (`char*`) is stored in the stack variable `pointer_storage`. Subsequently, the program reads user input into a 10-byte `name` array without specifying a maximum read length. As shown in Figure 4.1, the `name` array and `pointer_storage` are sequentially allocated on the stack. If the user input exceeds the limit of the `name` array, it triggers a buffer overflow, which could overwrite `pointer_storage`. As a result, an unintended string might be dereferenced and printed, presenting a vulnerability that an attacker could exploit. In the following sections, we will dissect the conversion processes to LLVM IR and WebAssembly code to understand how our design safeguards against such vulnerabilities.

LLVM IR Using our customized LLVM with Clang, we compiled the C code into the condensed LLVM IR snippet shown in Listing 4.11, discarding irrelevant lines and debugging metadata for clarity.

```
define i32 @__original_main() {
entry:
  %string = alloca ptr, align 8
  %pointer_storage = alloca ptr, align 8
  %name = alloca [10 x i8], align 1
  %loaded_string = alloca ptr, align 8
  store ptr @.str, ptr %string, align 8
  %string_signed = call ptr @llvm.wasm.pointer.sign(ptr %string)
  store ptr %string_signed, ptr %pointer_storage, align 8
  ...
  %1 = load ptr, ptr %pointer_storage, align 8
  %2 = call ptr @llvm.wasm.pointer.auth(ptr %1)
  %3 = load ptr, ptr %2, align 8 ; load after failed authentication
  store ptr %3, ptr %loaded_string, align 8
  %4 = load ptr, ptr %loaded_string, align 8
  %call4 = call i32 (ptr, ...) @printf(ptr @.str.4, ptr %4)
  ret i32 0
}
```

Listing 4.11: LLVM IR code example of PAC protection

The LLVM IR code starts by allocating pointers for the variables `string`, `pointer_storage`, `name`, and `loaded_string`. It then stores the initial string in the `string` pointer and signs the pointer using our custom function `llvm.wasm.pointer.sign`, before storing it in `pointer_storage`. After potentially causing a buffer overflow with the `scanf` function, which was omitted here for brevity, the code authenticates the PAC of the pointer loaded from `pointer_storage` and stores it in `loaded_string`. We can observe that if `pointer_storage` has been over-written and no longer stores the original signed pointer, the authentication will fail.

WebAssembly Code We have chosen not to include the actual WebAssembly code here due to its complexity, which does not offer any additional insight beyond the LLVM IR. For this somewhat contrived C example, enabling optimizations in LLVM would result in the complete removal of the `pointer_storage` due to its immutability, thereby eliminating the bug we aim to demonstrate. Thus, disabling optimizations preserves this aspect for demonstration purposes, but adds complexity to the generated WebAssembly code. Comprehending the LLVM IR should suffice as it offers the same understanding of where we insert our custom WebAssembly pointer instructions.

Execution in Wasmtime Since this program only displays a failure for specific user inputs, we can observe in the program output in Listing 4.12 that no runtime trap occurs in Wasmtime when the user input that is read using `scanf` is shorter than 10 bytes long.

This program does not raise a runtime error in Wasmtime when the user input read using `scanf` stays within the 10-byte limit, as shown in Listing 4.12.

```
What is your name?
123456789
Hello user 123456789!
String protected with PAC: Hello World!
```

Listing 4.12: Wasmtime executing PAC protected program without error

However, when the user input exceeds this limit, as depicted in Listing 4.13, the PAC protection mechanism detects an overwritten memory location due to the failure in authenticating the stored pointer. Despite recognizing the PAC trap, Wasmtime struggles to correctly associate the signal received from the OS with PAC due to an implementation detail, as elaborated in Section 5.1.2. Nevertheless, Wasmtime demonstrates reasonable accuracy in pinpointing the fault, although it is not as precise as with MTE traps.

```
What is your name?
1234567890
Hello user 1234567890!
Error: failed to run main module ‘demo.cwasm‘

Caused by:
    0: failed to invoke command default
    1: error while executing at wasm backtrace:
           0: 0x380 - main
                         at demo.c:13:27
           1: 0x26c - <unknown>!_start
    2: wasm trap: out of bounds memory access
```

Listing 4.13: Wasmtime error message for PAC protected program

5 Implementation

5.1 Wasmtime

5.1.1 Optimizing the Tagging of Memory Regions

Most of the enhancements required to support Memory Tagging Extensions (MTE) in a compiler are straightforward. For example, tagging pointers is relatively trivial, as this can be done manually using bit masks or with the `IRG` instruction. Since these instructions are lightweight, there is little room for performance optimization. However, the tagging of memory regions is a more complex process. Here, the AArch64 architecture provides us with instructions like `STG` for tagging individual memory regions. But since our aim is to tag multiple consecutive memory regions, we need to develop efficient algorithms for this purpose, thereby opening possibilities for performance improvements.

Naive algorithm

A naive algorithm for tagging memory regions that are referenced by an index is demonstrated in Listing 5.1. This algorithm iterates over the index in 16-byte steps, applying the `STG` instruction on each block. Before the loop, we need to ensure that the size is a multiple of 16 and that the iteration is feasible, i.e., `iter_ptr + size` does not overflow.

```
Function tag_memory_region(iter_ptr: i64, size: i64, tagged_ptr: i64):
    assert_16_byte_aligned(size)
    assert_no_overflow(iter_ptr + size)

    for (size; size >= 16; size -= 16, iter_ptr += 16):
        stg(tagged_ptr, iter_ptr)
```

Listing 5.1: Pseudo code for naive tagging algorithm

Optimization using `ST2G`

However, the aforementioned algorithm could become a performance bottleneck for larger values of `size`. To address this issue, we can leverage the `ST2G` instruction provided by the AArch64 architecture, which can tag a 32-byte memory region at once. This allows us to optimize our algorithm as shown in Listing 5.2. This optimization results in a noticeable improvement in performance, as discussed in Chapter 6.

```
Function tag_memory_region(iter_ptr: i64, size: i64, tagged_ptr: i64):
    assert_16_byte_aligned(size)
    assert_no_overflow(iter_ptr + size)

    for (size; size >= 32; size -= 32, iter_ptr += 32):
        st2g(tagged_ptr, iter_ptr)

    if (size == 16):
        stg(tagged_ptr, iter_ptr)
```

Listing 5.2: Pseudo code for tagging algorithm optimized with `ST2G`

Static and Dynamic Sizes

A critical distinction to consider is whether the `size` value is *static* or *dynamic*. A static value is known at compile time, perhaps because it is an integer literal in the source code (e.g., `int buffer[42]`). On the other hand, a dynamic value is only known at runtime (e.g., `int buffer[atoi(argv[1])]`). Depending on whether the size is identified as static or dynamic, different optimizations are applied.

Dynamic Size For dynamic sizes, our only feasible option is the implementation detailed in Listing 5.2, which utilizes `ST2G` to tag memory regions in 32-byte chunks, as the exact size of the region remains unknown at compile-time. This algorithm creates a loop where each iteration consists of a single `ST2G` instruction. Contrast this with an alternate approach where we have fewer loop total iterations, but each loop houses multiple `ST2G` instructions. Upon initial examination, these two methods seem equivalent performance-wise, as they would execute an identical total number of `ST2G` instructions at runtime. However, digging deeper uncovers the nuances that impact their actual efficiency. Our current implementation actually ends up executing a larger number of hardware instructions at runtime. This is primarily due to the runtime overhead associated with more branching instructions from the many loop iterations. In our approach, where the total number of loop iterations is relatively larger compared

to an approach with fewer loop iterations, a greater proportion of the execution time is spent on branching instructions. This inevitably leads to an increase in the overall number of instructions executed at runtime. To compound this issue, the frequent branching also introduces the hardware-specific issue of *branch misses*. A branch miss [29] happens when the processor's branch predictor fails to correctly guess the outcome of a branch instruction. When a branch miss occurs, the processor has to stall execution while it fetches the correct instructions, which can substantially slow down the overall execution time. As a result, even though our current implementation may appear equally efficient in the abstract sense, it can encounter performance challenges in real-world conditions due to the increased number of executed branching instructions and the potential for branch misses.

Static Size On the other hand, when the size is known at compile time, we can apply further optimizations. One such optimization is *loop unrolling* [11], a common compiler technique in which the loop structure is flattened, and the loop body is replicated, minimizing branching. In the context of our work, this would involve directly embedding all of the required `ST2G` and `STG` instructions, the required number of which we are able to calculate exactly by knowing the size at compile time, into the machine code without a loop. However, this approach is only feasible for small memory regions, as the code size would increase significantly with larger ones due to the duplication of instructions. To strike a balance between the benefits of loop unrolling and maintaining a smaller code size, we adopted a hybrid approach inspired by Clang's implementation for MTE. We introduced a *loop unrolling threshold*, representing the longest chain of `ST2G` instructions we would insert into the code, either directly without a loop (for total sizes smaller than the threshold) or within a loop body. Following empirical observations from Clang's established practice, we set this threshold to 160 bytes, equivalent to five `ST2G` instructions. This threshold strikes a balance between the performance gains from loop unrolling and the potential increase in code size. However, it is worth noting that the optimal threshold might vary depending on the specific workload and system characteristics, and tuning this parameter could be an avenue for further exploration. Given Clang's extensive resources and their expertise in compiler optimizations, we feel confident that their empirically determined threshold offers a robust starting point.

Potential improvements

One aspect we have yet to address is the possibility of an optimal tagging solution that does not require an explicit algorithm but instead uses a dedicated MTE instruction to tag an entire memory region as per the given size. Interestingly, the AArch64

Instruction Set Architecture (ISA) does include such an instruction known as `STGM` (Store Tag Multiple) [7], designed specifically for storing multiple tags in memory. Being a bulk tag manipulation instruction, `STGM` would be particularly suited for our needs. Unfortunately, `STGM` is undefined at Exception Level 0 (EL0) [6], where userspace level code is executed, thus preventing us from directly generating `STGM` instructions in our compiler. Despite its unavailability at EL0, `STGM` can be used at higher exception levels, such as Exception Level 1 (EL1), where the operating system kernel operates. This opens up the potential for implementing a system call that can be invoked from userspace. However, the viability of such an approach would depend on whether the performance gains from using the `STGM` instruction, invoked via a system call, would outweigh the potentially significant overhead of this system call. This trade-off can only be assessed through extensive performance testing. As our project does not extend to modifying the Linux Kernel, we have decided not to pursue this avenue. Furthermore, the fact that ARM did not define this instruction at EL0 suggests that, at least for the moment, compiler engineers have no alternative but to rely on `STG` and `ST2G`.

5.1.2 Trap Signals and Custom Error Messages

In previous sections, we have highlighted the promising capabilities of the ARM64 hardware extensions Memory Tagging Extension (MTE) and Pointer Authentication (PAC). Utilizing them to their full potential, however, necessitates integrated support within compilers and runtimes. Until recently, this was a functionality gap within Wasmtime.

MTE Our project has now successfully added comprehensive support for MTE within Wasmtime. Besides enabling the generation of new instructions related to this hardware feature when our custom WebAssembly instructions are detected in the input code, we have also addressed an equally significant issue: Identifying trap signals originating from MTE errors. This required enhancing Wasmtime's existing signal handling mechanism to differentiate MTE-induced faults from other types of errors, substantially improving the usability of MTE within Wasmtime and streamlining debugging.

In computer systems, signals provide a way for processes and the operating system to communicate, alerting the process about specific events [24]. In the context of Linux, a variety of signals exist, each linked to unique situations. For example, the `SIGINT` signal is sent when a user wishes to interrupt a process (usually with the `Ctrl+C` keybinding), whereas `SIGTERM` is the default signal sent to a process to terminate it.

A critical signal in the context of our discussion is `SIGSEGV`, which stands for *Segmentation Violation* and is sent to a process when it makes an invalid memory access.

In a broader context, `SIGSEGV` can occur due to a variety of reasons, such as a stack overflow, a null pointer dereference, and others.

To respond to these signals, processes use *signal handlers*. These are special functions defined within a process that are automatically invoked when the corresponding signal is received. These handlers enable the process to control its response to the event, which can range from ignoring the signal, performing clean-up tasks, to terminating itself.

Before our project's contributions, Wasmtime could differentiate between various types of `SIGSEGV` causes such as stack overflows or heap out-of-bounds errors, but it lacked the ability to recognize MTE-specific faults. To address this, we adapted Wasmtime to accurately identify and handle MTE-induced faults, guided by the *Tag Check Faults* section of the Linux Kernel Documentation [17].

In Linux, signals delivered to a process are accompanied by a `siginfo_t` [44] structure, offering additional information about the signal. A key field within this structure is `si_code`, which specifies the root cause of the signal. MTE introduces two such error codes, `SEGV_MTESERR` and `SEGV_MTEAERR`, signifying synchronous and asynchronous MTE errors respectively, derived directly from the Linux kernel's signal interface [26]. Our enhancement enables Wasmtime to extract this `si_code` when a `SIGSEGV` signal is received, which then allows the signal handler to identify an MTE error. We record the faulting address and the MTE fault status in a struct, which is then fed into Wasmtime's core WebAssembly trap handling mechanism. This mechanism processes the enriched metadata, distinguishes between MTE-related traps and other causes, and assigns a `MemoryTaggingExtensionFault` trap for detected MTE faults. Without an MTE fault, it resorts to the previous trap code assignment method.

It's worth noting that the information provided so far mostly applies to MTE's synchronous mode, where the signal along with the signal code are provided directly when the MTE trap occurs. However, MTE also offers an asynchronous mode where fault detection can be significantly more challenging, as the delay between the memory corruption and the fault can hinder the identification of the original context of the error.

PAC The situation with ARM64's Pointer Authentication (PAC) proves to be more complex. When a pointer, whose bits have been corrupted by PAC instructions, is dereferenced, a `SIGSEGV` signal is generated. However, unlike the synchronous MTE faults, this error occurs asynchronously, meaning the processor continues executing instructions until it dereferences the corrupted pointer, only then triggering the hardware exception. The lag between the initial pointer corruption and the eventual fault obscures the original context of the error, making it significantly harder to pinpoint and debug the issue. Unfortunately, neither the Linux Kernel documentation [41] nor

the ARM white paper [39] on PAC provides additional details on whether it is possible to differentiate a PAC-induced `SIGSEGV` signal from other `SIGSEGV` instances. Thus, at present, we have been unable to correctly identify Pointer Authentication hardware traps, leading Wasmtime to interpret them as conventional out-of-bounds accesses.

5.2 LLVM

5.2.1 Pointer Authentication Analysis

As mentioned in Section 4.2.1, our support for Pointer Authentication (PA) is based on an in-depth analysis that focuses on the problematic further uses and origins of memory locations containing pointers we aim to perform Pointer Authentication on. In this section, we will explain the implementation details of this analysis in LLVM, particularly the module pass, as it fundamentally extends the function pass, thereby incorporating all major elements of the function pass as well.

Our analysis in LLVM commences with the `runOnModule` function. In this function, we traverse all functions, collecting the store and load instructions of pointers we aim to protect by calling `authenticateStoredAndLoadedPointers`. We accumulate the instructions suitable for Pointer Authentication in a data structure, rather than directly inserting our PAC instructions. This approach is chosen to avoid active interference with the analysis while it is still in progress. Once all analyses for the module are completed, we separately insert the PAC instructions.

Our focus now shifts to `authenticateStoredAndLoadedPointers`, which is called by the `runOnModule` entry function and traverses all instructions. When it encounters a store or load instruction containing a pointer that passes the `memoryLocationIsSuitableForPA` check, it incorporates them into the returned data structure.

Next, we define the behavior of this function that checks whether a memory location in suitable for Pointer Authentication.

Definition 3 *A pointer, that is stored in or loaded from a memory location, is suitable for Pointer Authentication (PA) if and only if that memory location has* no other uses *and does not* come from elsewhere*. Moreover, a pointer is only suitable for PA if all of its aliases are also suitable for PA.*

The concept outlined in Definition 3 closely aligns with Requirement 3. Our implementation refines *received from external dependencies* to *comes from elsewhere*, and *provided to external dependencies* to *has other uses*. The way we implement this function is by marking a memory location as unsuitable for PA if it either has other uses or comes from elsewhere. For the sake of writing clear and comprehensible C++ code, these two

conditions are also implemented as functions `valueHasOtherUses` and `valueComesFrom Elsewhere`. To ensure comprehensive analysis, we extend this analysis to all aliases of a memory location, effectively iterating over all aliases (which conveniently includes the value itself).

Value Comes From Elsewhere

We can now explore `valueComesFromElsewhere` next, the behavior of which can be outlined as follows:

Definition 4 *A value "comes from elsewhere" if any of the following conditions are met:*

1. *The value was passed as a parameter to the current function.*
2. *The value was loaded from any memory location.*
3. *The value is a global value.*
4. *The value is the return value of any function.*

If the current value does not come from elsewhere, it is also necessary to check whether any of its operands originate from elsewhere, because a value can also transitively come from elsewhere.

In cases where the value is a parameter that is passed to the current function, we reach a limitation in tracing its origin. Since we lack knowledge about the current function's possible callers, we must classify the value as originating from elsewhere. This approach mirrors our handling of a memory location loaded from another location we're analyzing - again, we simply cannot extend our analysis further. Global values also present a challenge. By their nature, these values are accessible throughout any module in LLVM. Due to their global reach, we must categorize them as coming from elsewhere. A similar concern arises with function return values. Because we cannot reliably trace the internal behavior of every function, we are unable to perform Pointer Authentication on a memory location that is a function's return value.

Of equal importance is our consideration of all other operands, similar to our analysis of aliases. In LLVM, instructions are represented as objects with operands and a resultant value. An operand can be thought of as an input to an instruction, such as a variable, constant, or output of another instruction. For instance, consider an addition instruction, `c = a + b`. Here, `a` and `b` are the operands, while `c` is the resultant value. Take for example a memory location resulting from the addition of an offset to another memory location that indeed originates from elsewhere. In this scenario, the offset and the other memory location are the operands for our subject memory location. This example underlines the importance of verifying whether any operands originate from

elsewhere. Otherwise, we risk overlooking situations where Pointer Authentication should not be performed on a memory location.

Note that it is possible that the rules specified in Definition 4 may be stricter than necessary, but they outline our current working implementation. It is important to clarify that we are not asserting that a value conclusively originates from an external dependency each time our analysis determines that it comes elsewhere. In the interest of caution, we might categorize items as externally originating even when they may not be, due to the limitations of our analysis. Thus, when our analysis concludes that a value comes from elsewhere, it implies we cannot guarantee that it does not come from an external dependency. The rationale behind our cautious approach is as follows: It is tolerable to miss some edge cases where we fail to apply Pointer Authentication on a pointer, even if it would be feasible - while this does not enhance the generated output, it does not degrade it either. However, if we mistakenly apply Pointer Authentication to a pointer that should be disregarded, this could lead to a PAC-triggered system crash, causing our solution to inflict more harm than good.

Value Has Other Uses

Our attention now shifts to the function `valueHasOtherUses`, which operates as detailed below:

Definition 5 *A value has other uses if it is recursively passed as a function parameter to an external function. Furthermore, even if a value is passed to a non-external function, it is necessary to verify if the value has other uses within that function. A value also has other uses if any of its aliases has other uses.*

This function primarily depends on its helper function, `findAllFunctionsWhereValueIsPassedAsArgument`, which identifies all functions that are passed the value as an argument. Notably, this analysis must also include all aliases of the value. If any of these functions are external (as per Definition 2), the value indeed has other uses, precluding the use of Pointer Authentication.

Pseudo code for `findAllFunctionsWhereValueIsPassedAsArgument` is presented below in Listing 5.3.

It begins by identifying all functions to which our value is recursively passed as an argument. The recursive nature of this implementation lies in the fact that once we have established our value being passed to a particular function, we must extend our analysis to check if that value has other uses within in that other function. Notably, if the function that the value is passed to is the same function we are currently analyzing, meaning we have encountered a recursive function call in the input source code, we skip analyzing the passed to function further. However, there are also cases where

```
find_functions(value, functions):
    for user in value.users:
        if call_base = call_base(user):
            for argument in call_base:
                if call_base == value:
                    function = call_base
                    if unknown_function_signature(function):
                        return IsExternalFunction

                    functions.add(function)

                    if function == cur_function:
                        continue

                    if hasOtherUses(function.get_arg(value), function):
                        return IsExternalFunction

        if find_functions(user, functions) == IsExternalFunction:
            return IsExternalFunction

    return IsNotExternalFunction
```

Listing 5.3: Pseudo code for `findAllFunctionsWhereValueIsPassedAsArgument`

our value is passed to a function pointer or vararg function. These function calls are impossible to analyze because their characteristics are not known at compile time. In such scenarios, our analysis opts for caution, and terminates early, indicating that PA cannot be applied.

6 Evaluation

In this section, we evaluate our research question to determine whether we can achieve memory safety in WebAssembly without significant performance compromises.

6.1 Testbed

Due to the absence of available ARM hardware that supports the Memory Tagging Extension (MTE), we turned to QEMU for benchmarking. While this may not provide exact real-world performance metrics for prospective ARM hardware with MTE support, it offers invaluable relative comparisons. Such comparisons assist us in assessing the effectiveness of our optimization efforts and in identifying potential bottlenecks. By analyzing various C source programs, we aim to confirm and extend the list of tradeoffs we detailed in Section 2.4.2 on ideal input programs that maximize the performance with our toolchain.

Our benchmarking was performed on a high-end Linux computing server equipped with an AMD EPYC 7713P CPU with 64 physical cores and 128 threads, along with 515 GiB of DDR4 RAM, running NixOS 23.05. Despite its robust multithreading capabilities, our benchmarks utilize only a single core. This choice was influenced by WebAssembly's inherent single-threaded nature, and our desire to prevent potential thermal throttling which could affect results. We opted to run all benchmarks sequentially. Additionally, benchmarking was conducted during periods of minimal server load.

The cornerstone of our setup is the QEMU emulator (version 6.0.0). QEMU requires emulating an entire operating system when enabling MTE, which means we had to use the `qemu-system-aarch64` configuration with the `-M mte=on` option.

Given our lack of access to devices supporting Pointer Authentication for benchmarking, PAC benchmarks were also run in QEMU.

To measure CPU cycle counts for each Wasmtime invocation, we utilized the Linux `perf` program, treating these cycles as a program's runtime.

6.2 Overhead Assumptions

We begin by pinpointing potential overhead-contributing factors in our implementation that might impact our benchmarks. Recognizing these factors will later facilitate better understanding of the observed overhead sources in our benchmarks.

- *Execution of MTE Tagging Instructions*
  This overhead stems from executing the MTE-related AArch64 instructions we generate in Wasmtime (`IRG`, `STG`, and `ST2G`) on the hardware (QEMU). The performance is currently dependent on QEMU's implementation, rendering a direct performance comparison to non-MTE programs less meaningful than contrasting different tagging strategies.

- *Execution of Pointer Authentication Instructions*
  Similar to MTE, this overhead originates from executing the PAC-related AArch64 instructions (`PACDZA` and `AUTDZA`) in QEMU. Again, the incurred overhead is entirely dependent on QEMU.

- *Execution of Memory Access Instructions with MTE*
  This overhead arises not from the execution of specific new MTE instructions, but from enabling MTE on the target system. Every memory access with MTE enabled necessitates a tag comparison between the memory address and location. This factor remains outside our direct control, and also solely dependent on QEMU.

- *MTE Synchronous Mode*
  As we have detailed before, MTE supports both a synchronous and asynchronous mode, with the asynchronous mode promising improved performance at the cost of diagnostic information. Once more, we do not know how this feature is implemented inside QEMU, and whether its performance reflects that of real hardware.

- *MTE Infrastructure in LLVM*
  Unlike PAC-related LLVM passes, the MTE LLVM pass also mandates size-tracking of memory regions to facilitate their de-allocation when obsolete. The current data structure for this purpose is a HashMap, which might become a performance bottleneck with frequent heap allocations.

6.3 Methodology

Our benchmarks span various combinations of the features and optimizations we have introduced.

The combinations at the LLVM level are straightforward. We detail the distinct WASM file variants generated by LLVM, which are then used by various Wasmtime configurations.

- *None*
  No custom WASM instructions are introduced.
- *MTE Only*
  Only MTE-specific (`segment.new`, `segment.free`) WASM instructions are inserted.
- *PAC Only*
  Only PAC-specific (`i64.pointer_sign`, `i64.pointer_auth`) WASM instructions are inserted.

Unfortunately, supporting MTE and PAC together in the current prototype does not yield the expected results. This is because arrays that contain pointers cannot be protected by PAC in our current implementation, since the arrays (memory locations) are classified as coming from the external function `segment.new`. This is a weakness of our current implementation, and should be addressed in future work.

In Wasmtime, we have devised more intricate optimizations for examination. Thus, we enumerate these Wasmtime configurations:

- *No MTE*
  This benchmark utilizes the *None* WASM code and sets a baseline against which other configurations are normalized.
- *MTE Infrastructure Only*
  Although this variant uses the *MTE Only* WASM code, leading LLVM to generate code for an MTE-enabled system, MTE is fully disabled within Wasmtime. Any MTE-specific instructions in the input WASM code are treated as NOPs. This approach helps evaluate the overhead from non-MTE infrastructure, such as our HashMap.
- *No Tagging Instructions*
  This configuration uses the *MTE Only* WASM code, activates MTE in Wasmtime but does not perform any memory tagging. No `IRG`, `STG`, or `ST2G` instructions are inserted, even though MTE is activated within the Linux environment, resulting

in all addresses being tagged with the default zero tag. The intent is to evaluate the overhead contributions from our different memory tagging algorithms.

- *Naive STG*
  This variant fully utilizes our MTE additions in Wasmtime using the *MTE Only* WASM code, but employs only the basic algorithm for all segment tagging.
- *Optimized ST2G*
  This configuration is similar to the previous variant, but it makes use of the enhanced `ST2G` tagging algorithm that we introduced in Section 5.1.1 for all segment tagging.
- *Optimized ST2G + Loop Unrolling Threshold*
  This variant uses the `ST2G`-optimized tagging algorithm for dynamically sized segments, but the loop unrolling threshold optimization for statically sized segments.
- *MTE Async Mode + All Optimizations*
  All variants detailed so far use MTE's synchronous mode. This variant includes all optimizations from the previous variant, but uses the asynchronous MTE mode.
- *PAC Only*
  This sole Wasmtime variant for PAC leverages the *PAC Only* WASM code and generates `PACDZA` and `AUTDZA` instructions.

6.4 Benchmarks

To evaluate the efficiency of our implementation, we have selected an assortment of C source code, encompassing different computing tasks and requirements.

6.4.1 Sorting Algorithms

Our first benchmarking focus was on comparing the runtime of various sorting algorithms on an array of integers of size 40,000. Specifically, we tested bubble sort, merge sort, and a custom-designed modified merge sort. These algorithms were chosen because of their contrasting time complexities - $O(n^2)$ for bubble sort and $O(n \log n)$ for merge sort - and auxiliary space complexities - in-place with $O(1)$ space for bubble sort versus $O(n)$ for merge sort.

The regular merge sort performs all allocations on the heap using `malloc`. In contrast, the modified version utilizes a distinct strategy for merging two sorted sections, as

shown in Listing 6.1. When merging partitions of a size less than a predefined *LIMIT*, this version allocates buffer space on the stack rather than the heap. Although this may seem inefficient due to the constant allocation of LIMIT-sized arrays, this memory overhead is deliberate. A deeper analysis of our rationale and the chosen LIMIT value will be presented later in the benchmark result discussions in Section 6.5.1.

```
void merge(int* arr, size_t l, size_t m, size_t r) {
    size_t L_size = m - l + 1, R_size = r - m;

    if (L_size <= LIMIT && R_size <= LIMIT) {
        int L[LIMIT], R[LIMIT];
        // Copy data to temp arrays L[] and R[]
        perform_merge(arr, L, R, l, L_size, R_size);
    } else {
        int *L = (int *) malloc(L_size * sizeof(int));
        int *R = (int *) malloc(R_size * sizeof(int));
        // Copy data to temp arrays L[] and R[]
        perform_merge(arr, L, R, l, L_size, R_size);
        free(L);
        free(R);
    }
}
```

Listing 6.1: Merging of two partitions in the modified merge sort

6.4.2 Static-Size Allocations

Next, we delve into the performance variations between different memory tagging techniques when handling large, stack-allocated arrays of static size. For an array to have a static size known at compile-time, it must be allocated on the stack.

At first, we intended to utilize C code for this benchmark. However, we observed that LLVM automatically adds extra instructions to guarantee proper alignment for stack-allocated arrays with static sizes. This causes the size passes to our `segment.new` to lose its static nature. Therefore, we manually wrote LLVM IR code. In this code, a large array is allocated on the stack, and the `segment.new` and `segment.free` instructions, which both individually initiate the tagging of the entire array, are executed in a loop. To evaluate the overhead implications, we adjusted the array size and loop iteration count, selecting diverse combinations for our benchmarks. Given the limited default stack size in C (around 8 KiB, sufficient for at most 2,000 integers), we used LLVM

compiler flags to extend the C stack size and WASM linear memory size to better test the memory tagging algorithms.

6.4.3 Pointer Authentication

Unfortunately, the sorting algorithms discussed earlier do not store pointers to or load pointers from any memory locations. Hence, we devised an additional stress test for our PAC implementation that is illustrated in Listing 6.2. This test involves the allocation of a large pointer array on the stack, storing, and subsequently loading pointer values. To facilitate the use of optimization flags (`-O2`), we integrated extra logic into this test that prevents the loops from being optimized away. Analogous to Section 6.4.2, adjustments were made to the C stack and linear memory sizes to avoid stackoverflows.

```
int test(size_t n) {
    void* ptrArray[n];
    size_t sum = 0;
    for (size_t i = 0; i < n; i++) {
        ptrArray[i] = (void*) i;
    }
    for (size_t i = 0; i < n; i++) {
        sum += (size_t) ptrArray[i];
    }
    return sum % 125;
}
```

Listing 6.2: Storing and loading pointers in loops

6.5 Results

To refine our results and ensure accuracy, each benchmark was executed five times, and we then derived the mean and standard deviation from these outcomes.

6.5.1 Sorting Algorithms

Insights from Figure 6.1 allow us to make several observations.

MTE Infrastructure Only

What immediately stands out is that the *MTE Infrastructure Only* Wasmtime variant introduces an overhead exceeding one order of magnitude (3735%) for the merge sort

algorithm, while this overhead is only 1% for bubble sort and 10% for modified merge sort. The underlying reason for this discrepancy is most likely the *MTE Infrastructure in LLVM*. This factor is more pronounced in the merge sort algorithm because this algorithm has more heap allocations and de-allocations with `malloc` and `free` compared to bubble sort. We must remember that each allocation and de-allocation requires traversing the HashMap that stores the pointers and the allocated sizes. The bubble sort algorithm does not perform any significant allocations besides that of the unsorted input array, because it is an in-place algorithm. Meanwhile, the modified merge sort has the same amount of total allocations as the regular merge sort, but most of these allocations are on the stack, shielding them from the LLVM infrastructure. The relatively small 10% overhead of the *MTE Instrastructure Only* variant for the modified merge sort algorithm likely corresponds to the heap allocations during the merging of partitions larger than the *LIMIT* constant. From these observations, we can conclude:

- The tradeoff for frequent allocation and de-allocation is pronounced for our toolchain, with heap-based allocations incurring a higher overhead than stack-based ones. Thus, for performance improvement, we suggest using the stack for frequent, small allocations.
- The overhead of the LLVM infrastructure identifies significant potential for future optimizations, discussed further in Chapter 9.

No Tagging Instructions

The *No Tagging Instructions* variant reveals intriguing overheads, ranging from a mere 81% for the modified merge sort to a significant order of magnitude (1206%) for bubble sort. This variance likely emerges because the algorithms' time complexity gets amplified by overheads associated with memory access instructions in MTE-enabled systems. As every store and load instruction in MTE systems incurs overheads (due to tag comparisons), the quadratic time complexity of bubble sort amplifies this overhead. We cannot necessarily influence or reduce this overhead in our implementation, since it is due the to inherent nature of MTE itself. However, this overhead might also not be representative of the real-world performance of MTE-enabled hardware, since we are running in the emulated QEMU environment. Additionally, these results stem from allocation or memory access-heavy programs. We estimate that compute-heavy programs would experience a considerably reduced runtime overhead.

Tagging Instructions

Finally, we can interpret the differences between the *No Tagging Instructions* variant and the other three variants that perform memory tagging on the segments. We notice that

the *Naive STG*, *Optimized ST2G* and *Optimized ST2G + Loop Unrolling Threshold* variants are quite close to the *No Tagging Instructions* variant for bubble sort and merge sort. For bubble sort, this is because we only perform memory tagging exactly twice, once when allocating and once when de-allocating the input array that is to be sorted. For merge sort, the runtime overhead is dominated by excessive heap allocations, so any further overhead by memory tagging does not significantly influence the result. However, the results observed in the modified merge sort demonstrate why we chose to add this variant in the first place. The modified merge sort shows an overhead of at least 22% for all three tagging algorithms compared to the *No Tagging Instructions* variant. This overhead arises because we tag more memory than is strictly necessary for a merge sort implementation. As shown earlier in Listing 6.1, even for small partitions a buffer of static size *LIMIT* is allocated on the stack.

Our decision to incorporate the modified merge sort algorithm in our benchmarks had the following underlying motives:

- **Heap-based Allocations Bottleneck:** Our primary aim was to validate whether multiple heap-based allocations present a performance bottleneck in our system. This was confirmed in the preceding discussion. For a direct comparison, it was crucial to replace a significant number of heap-based allocations with stack-based counterparts, keeping other elements unchanged. It should be noted, however, that the mere transition from heap to stack allocations did not necessitate us to define a static size for each stack allocation.
- **Loop Unrolling Threshold Optimization:** An additional goal was to design a scenario where buffer sizes are statically known at compile-time, allowing us to evaluate our loop unrolling threshold optimization. This motivation explains our choice of 160 bytes as the *LIMIT*, since this value represents the upper limit where our optimization fully unrolls the `ST2G` loop. Hypothetically, under these conditions, our optimization should perform better than the regular `ST2G` optimization.

However, the observed outcomes slightly diverged from our expectations. The *Optimized ST2G* variant only has 2% greater overhead compared to the loop unrolling threshold variant. We speculate this minor performance gap might stem from the relatively small size of the tagged segments, as it seems the optimization might not have been fully exploited.

6.5.2 Static-Size Allocations

Figure 6.2 allows us to compare the runtimes of various memory tagging approaches.

Initially, for a small count of executed memory tagging extensions, the variance between the different tagging strategies appears minimal, likely due to other dominant runtime overheads, such as the startup time of the WASM runtime. As the total bytes tagged (considering both loop iterations and segment size) increases, the optimized methods achieve close to half the runtime of the naive strategy. This is aligns with our expectations because the `ST2G` instruction in the optimized versions tags two 16-byte granules, in contrast to the `STG`'s single granule, necessitating only half the `ST2G` instructions.

It is worth noting that as the array size grows, the relative startup overhead of the WASM runtime diminishes. This is because the cost of the tagging instructions begins to overshadow other operations, making them the primary factor influencing runtime.

More intriguingly, the added loop unrolling threshold optimization does not visibly enhance performance. A possible explanation is the dominance of the `STG` and `ST2G` instructions in runtime over loop instructions. This implies that these tagging instructions might demand a significantly larger CPU share than the loop instructions. We initially assumed that partially unrolling the `ST2G` loop up to a certain threshold would reduce overhead from loop instructions while maintaining a similar binary size. Yet, it remains uncertain if this outcome stems from QEMU's specific implementation or genuinely mirrors real hardware performance.

Binary Size Overhead It is noteworthy that we measured the binary size overhead due to the two optimization strategies to be minimal compared to the naive algorithm or the Wasmtime variant without MTE. In general, the binary size overhead for heap-based allocations remains constant as tagging instructions are integrated only into the `malloc` and `free` functions. However, for stack-based allocations, the overhead grows linearly as tagging instructions are added for each stack-based allocation.

MTE Asynchronous Mode

Unfortunately, Figure 6.1 suggests that activating MTE's asynchronous mode did not yield additional performance enhancements. However, this observation only increases the likelihood of our assumption that QEMU's implementation of MTE might not fully align with its real-world hardware counterpart, especially given the performance promises associated with the asynchronous mode.

6.5.3 PAC

Figure 6.3 suggests that the relationship between array size and PAC overhead hovers somewhere between linear and quadratic. Building on earlier discussions, it is worth

noting that the use of QEMU might influence these results, and actual hardware may perform significantly better.

But even if PAC introduces overhead, it is critical to understand these findings in the context of real-world scenarios. In practice, vast arrays are rarely allocated on the stack. Instead, the heap is a more conventional destination for such allocations. Therefore, it is worth clarifying why we opted to allocate these extensive arrays on the stack rather than the heap in our tests. Our current LLVM module pass cannot protect pointers stored in heap memory locations. The reason being, the heap-allocated array is deemed as *coming from elsewhere*, given that it originates from the external `malloc` function. One might interpret this as a limitation in our implementation's optimal safety potential, which future iterations using Link-Time-Optimization (LTO) (see Chapter 9) might aim to mitigate. However, this could also be seen as our strategy of prioritizing the protection of stack-based allocations over heap-based ones, given the greater vulnerability of the former to malicious exploits. And given that, in typical C applications, allocations on the stack are limited in size, the overhead introduced by our PAC mechanism should remain minimal in real-world scenarios.

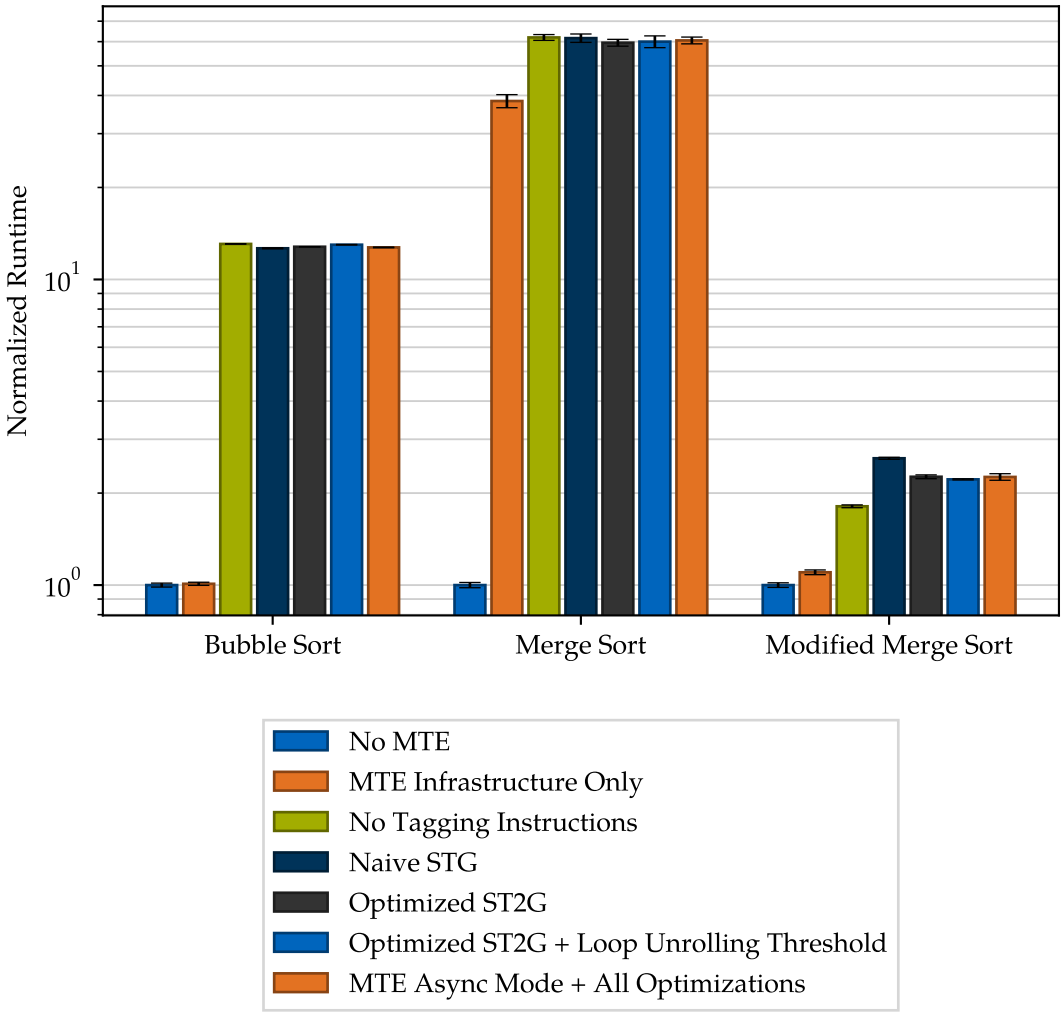


Figure 6.1: Runtime overhead of sorting algorithm benchmarks

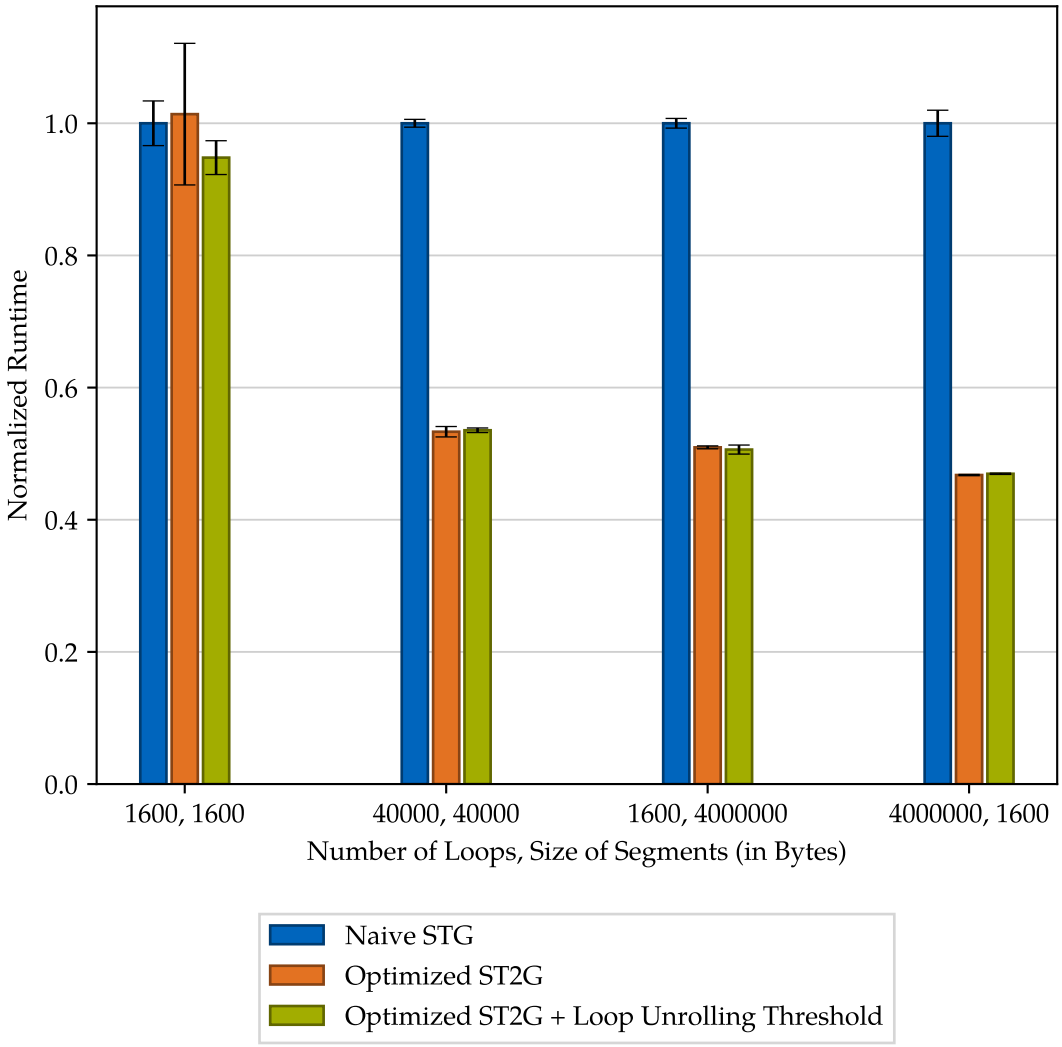


Figure 6.2: Runtime overhead of different memory tagging strategies

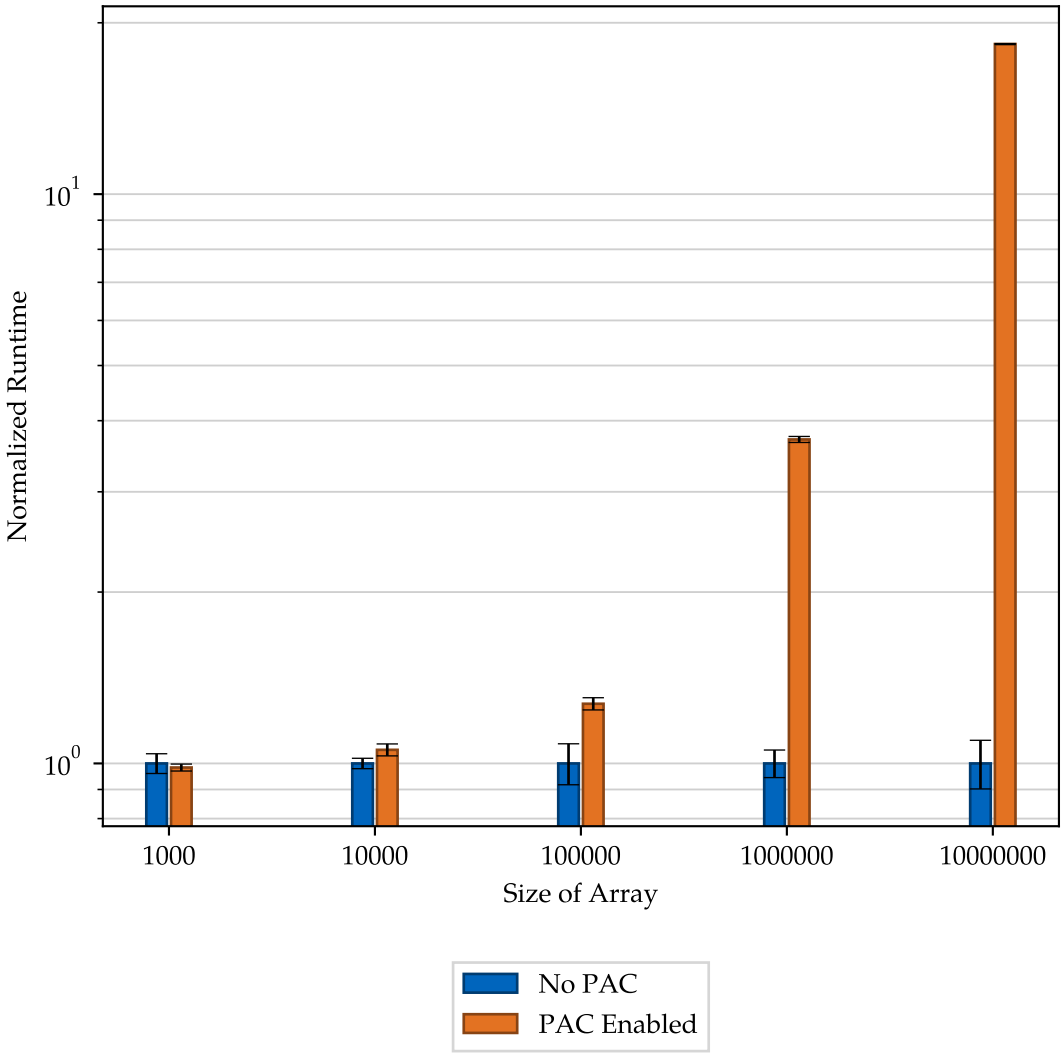


Figure 6.3: Runtime overhead PAC instructions

7 Related Work

Our project merges two primary ideas: First, ensuring memory safety *in WebAssembly*. Second, leveraging *hardware extensions* for enhanced memory safety. While our approach to combining these ideas is somewhat unique, there has been significant research on each of them individually.

7.1 Providing Memory Safety in WebAssembly

In this section, we cover related work that explores different approaches to ensuring memory safety in WebAssembly.

7.1.1 Memory-Safe WebAssembly (MSWasm)

Memory-Safe WebAssembly (MSWasm) [30] is an extension of WebAssembly that provides enhanced memory safety features. MSWasm's defense mechanisms have been tailored to defend against a WASM-level attacker who targets memory vulnerabilities in C programs compiled to Wasm. This protection spans across spatial memory errors such as buffer overflows, temporal errors like use-after-free, and violations of pointer integrity. They give a precise and formal semantics of MSWasm, and prove that well-typed MSWasm programs are inherently memory safe.

To ensure memory safety, MSWasm uses *segments* — regions of the linear memory, which provide a protective layer against unsafe memory accesses. Unlike standard WebAssembly linear memory, these segments are not directly accessible via typical load and store operations. Instead, they are accessible only through specialized constructs called *handles*. These handles are tuples of the form ⟨`base, offset, bound, isCorrupted, id`⟩. They effectively act as unforgeable capabilities, binding pointers to specific segment allocations. The *base* points to the segment's starting point, while the *offset* indicates its position within the segment. Together, they determine the exact address being pointed to. The *isCorrupted* flag ensures handle integrity. Any attempts to forge these handles, like manipulating their bit representation, result in a corrupted handle. Interestingly, the WASM runtime, in which MSWasm is executed, does not immediately trap when a corrupted handle is created. Instead, it traps only when such a handle is used, enhancing performance by removing checks on every pointer

arithmetic operation. Moreover, each segment allocation is associated with a unique *id* for enforcing temporal memory safety.

MSWasm also introduces specific instructions tailored for these segments and handles. While `segload` and `segstore` resemble standard load and store operations, they operate strictly on handles. They trap if a handle is corrupted, if it references outside the segment bounds, or if the segment it points to is already freed. Segments are managed with instructions like `segalloc` and `segfree` for allocation and de-allocation, respectively. The `handle.add` instruction facilitates pointer arithmetic, tweaking the handle's offset but not the base or bound.

Besides its design, an interesting point is also MSWasm's actual implementation. A major difference to our project is that MSWasm does not directly encode the handle's capabilities in the pointer itself (like we do by using MTE), but instead leverages the concept of *fat pointers*. Fat pointers are extended pointer structures which carry additional metadata about the memory they point to. This also explains why MSWasm decided to use the CHERI [15] LLVM fork that includes support for fat pointers. Part of their efforts was to lower CHERI's fat pointer abstractions to the MSWasm handle abstractions. Just like WebAssembly, MSWasm is designed as a target for higher-level languages, specifically C in this context. Their extensions to the CHERI compiler guarantees that C programs, when compiled into MSWasm, either retain their safe attributes or trap at the very first instance of memory violation, hence ensuring safety.

One can easily recognize the similarities between MSWasm and our project, primarily because MSWasm served as a significant inspiration for our work. Both initiatives leverage LLVM (upon which CHERI is based) and WASI-libc, share the concept of segments, and utilize additional metadata bits for detecting memory safety issues. While our overarching strategy and objectives align with MSWasm, we have chosen hardware-level extensions, namely MTE and PAC, to eliminate the need of for intricate software-based handle structures. This approach, though constrained to specific hardware, offers promising potential for minimized performance overhead. It is evident that MSWasm also recognizes the advantages of hardware extensions, as highlighted by their plans to incorporate MTE into their system as well in the future.

7.1.2 WASM proposal on built-in Garbage Collection

Garbage Collection (GC) is a pivotal feature in modern programming languages, facilitating automated memory management by freeing space occupied by objects no longer in use. Although GC offers guaranteed memory safety, it entails a significant runtime overhead due to continuous checks for reclaiming unused memory.

Presently, WebAssembly lacks native garbage collection. This means that GC-based programming languages have to either incorporate their own garbage collector or

compile their entire runtime, including the garbage collector, to WebAssembly. Both strategies result in increased binary sizes, leading to prolonged download and startup durations. [18]

However, there is an ongoing proposal [40] to integrate garbage collection directly into WebAssembly. While source languages that are not garbage-collected, like the C language we focus on, will not benefit from this, it represents an interesting alternative approach to providing memory safety in WebAssembly. Nevertheless, once WebAssembly's garbage collection feature is rolled out and as MTE-enabled hardware becomes more prevalent, a performance comparison between both methods will be compelling, albeit for different source languages.

7.2 Providing Memory Safety with ARM64 Hardware Extensions

In this section, we delve into related work that investigates methods for achieving memory safety utilizing ARM64 hardware extensions.

7.2.1 Hardware-assisted AddressSanitizer

Google's Hardware-assisted AddressSanitizer (HWASan) [3, 42] is another compelling approach to the detection of memory safety issues. Available on Android 10 devices with AArch64 hardware, HWASan identifies a variety of memory safety concerns similar to our project. Instead of leveraging the Memory Tagging Extension (MTE), HWASan relies on software-defined tags and utilizes the Top Byte Ignore (TBI) feature for tag storage. The process also involves a combination of more general instructions for operations like random tag generation, pointer tag insertion, and memory region tagging. This method, however, results in a notable code size overhead due to the reliance on many general-purpose instructions. In contrast, MTE accomplishes similar tasks with specialized instructions like `IRG` and `STG`. Our approach to detecting heap-related memory issues draws inspiration from this paper, especially their detailed modifications to `malloc` and `free`, which outline steps such as allocation alignment by a tag granule (TG), random tag selection, memory tagging for allocated chunks, and address return with tags [42].

7.2.2 Deterministic Tagging for Memory Safety

Liljestrand et al. [25] present an intriguing method for memory tagging in their LLVM extension, which takes full advantage of ARM's Memory Tagging Extension (MTE).

Their system differentiates between safe and unsafe memory allocations using a deeper LLVM analysis. This distinction permits the tagging only of those memory allocations deemed unsafe by their analysis, thus reducing the overall system overhead. By contrast, our toolchain, in its current form, tags every allocation under the assumption that they are potentially unsafe.

The central idea detailed in their study is to categorize safe allocations into three distinct types: The first type includes allocations always considered safe due to being dereferenced solely by compiler-generated code. The second type encompasses those allocations that the analysis can conclusively prove to be safe due to all pointer dereferences based on the allocations being proven safe. Lastly, there are guarded allocations, for which safety cannot be entirely guaranteed, but any unsafe dereferences can be identified and addressed using their instrumentation. This approach allows the system to focus security mechanisms precisely where they're needed, thus maximizing efficiency.

In their benchmarks, they found that an impressive 41% of all allocations were determined to be safe, eliminating the need for MTE-based protection for these allocations. This significant optimization not only underscores the effectiveness of their analysis but also highlights the promising potential it offers for other memory safety endeavors.

8 Conclusion

In our work, we extended WebAssembly, LLVM, and Wasmtime to develop a functional prototype toolchain. Using a selection of C programs with known vulnerabilities, we successfully leveraged the ARM64 hardware extensions to pinpoint both spatial and temporal memory safety issues within WASM programs. Additionally, our enhancements to Wasmtime error messages for identifying MTE traps enrich the debugging experience within our toolchain. Our evaluations confirmed that most of our optimization strategies offered performance gains compared to more basic algorithms.

However, it is essential to note that our benchmarks, conducted within the emulated QEMU environment, did present a noticeable performance overhead. While this overhead is less critical for PAC, given its more niche application scenarios, it becomes a pronounced concern for MTE, which sees a wider use. However, it is evident that much of this overhead is tied to QEMU's current MTE and PAC implementations. The same performance being observed between MTE's synchronous and asynchronous modes in QEMU, even though the latter is expected to be superior, further supports this. Consequently, definitive performance insights can only be derived from tests on real hardware in the future.

With these considerations in mind, our outlook remains optimistic. We anticipate more favorable performance metrics, particularly for MTE, when assessments are conducted with real hardware support. The preliminary indicators are encouraging, leading us to envision a future where WebAssembly, protected by ARM's hardware capabilities, serves as an even more valuable tool for developers.

9 Future Work

We have previously already identified a significant overhead caused by the MTE infrastructure in LLVM. This arises when LLVM needs mechanisms to allow our `free` wrapper functions to know the size of the memory region to untag. Instead of storing these sizes in a HashMap, which results in runtime overhead with each access, we propose a direct memory overhead approach. By allocating slightly more memory than is requested during the allocation process, we can store the size of the memory region directly within this extra memory. This approach seems consistent with MTE's principles, which balances slight memory overhead for improved memory safety and runtime performance.

Another area of improvement is expanding the Pointer Authentication analysis in LLVM to cover the whole input program, not just isolated modules. This would allow us to label functions as external only when they are truly outside the project scope, extending protection for pointers in memory. Our strategy for this is to incorporate a Link Time Optimization (LTO) pass. This happens during the linking phase, after all source files are compiled. At this point, LLVM can see all translation units, enabling optimizations or transformations across them.

At present, our use of MTE for memory segment protection contains a degree of non-determinism. Specifically, there is a possibility that consecutive memory segments might receive identical tags since the `IRG` operation can yield the same tag. To address this, we are considering an approach where we can deterministically guarantee that two consecutive segments are never assigned the same tag. The proposal involves utilizing the tag produced by `IRG` for the initial segment, followed by successive increments of that tag for subsequent segments.

Lastly, we aim to make our WebAssembly contributions backward compatible. While we understand that getting our new WASM instructions into the official WebAssembly standard might be challenging at this time, we nevertheless want WASM binaries created with our memory safety features to work on other WebAssembly runtimes, even those not familiar with these instructions. The exact implementation remains a work in progress, marking another direction for future work on the project.

Abbreviations

List of Figures

List of Tables

Bibliography

[1] A. A. de Amorim, C. Hritcu, and B. C. Pierce. *The Meaning of Memory Safety*. 2018. arXiv: 1705.07354 [cs.PL].

[2] Android. *Arm Memory Tagging Extension (MTE): Understanding MTE reports*. 2023. URL: https://source.android.com/docs/security/test/memory-safety/mte-reports (visited on 08/04/2023).

[3] Android. *HWAddressSanitizer*. 2023. URL: https://source.android.com/docs/security/test/hwasan (visited on 08/14/2023).

[4] Apple Inc. *Preparing Your App to Work with Pointer Authentication*. URL: https://developer.apple.com/documentation/security/preparing_your_app_to_work_with_pointer_authentication (visited on 08/04/2023).

[5] *ARM*. URL: https://www.arm.com/ (visited on 08/04/2023).

[6] ARM Limited. *Armv8-A Architecture Concepts: Exception Levels*. URL: https://developer.arm.com/documentation/ddi0488/d/programmers-model/armv8-architecture-concepts/exception-levels (visited on 08/04/2023).

[7] ARM Limited. *ARMv8-A Architecture Reference Manual: Base Instructions*. 2023. URL: https://developer.arm.com/documentation/ddi0602/2023-06/Base-Instructions (visited on 08/04/2023).

[8] ARM Limited. *Armv8.5-A Memory Tagging Extension*. 2023. URL: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf (visited on 08/04/2023).

[9] S. Bibi, M. Asghar, M. U. Chaudhry, N. Hussan, and M. Yasir. "A Review of ARM Processor Architecture History, Progress and Applications." In: 10 (Aug. 2021), pp. 171–179.

[10] Z. Blazer. *History of the evolution of the x86 platform, from the IBM PC to the modern era*. Version 01/01/2021. 2021. URL: https://zirblazer.github.io/htmlfiles/pc_evolution.html (visited on 08/14/2023).

[11] M. Booshehri, A. Malekpour, and P. Luksch. *An Improving Method for Loop Unrolling*. 2013. arXiv: 1308.0698 [cs.PL].

[12] Bytecode Alliance. *Wasmtime: A fast and secure runtime for WebAssembly*. URL: https://wasmtime.dev/ (visited on 08/04/2023).

[13] Z. Chen, C. Tao, Z. Zhang, and Z. Yang. "Beyond Spatial and Temporal Memory Safety." In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ICSE '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 189–190. ISBN: 9781450356633. DOI: 10.1145/3183440.3195090.

[14] L. Clark. *Standardizing WASI: A system interface to run WebAssembly outside the web*. URL: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/ (visited on 08/04/2023).

[15] CTSRD CHERI. *The CHERI LLVM Compiler Infrastructure*. URL: https://github.com/CTSRD-CHERI/llvm-project (visited on 08/13/2023).

[16] C. Foster. "Computer Architecture." In: *Computer* 5.2 (1972), pp. 18–19. DOI: 10.1109/C-M.1972.216888.

[17] V. Frascino and C. Marinas. *Memory Tagging Extension (MTE) in AArch64 Linux*. Technical Report. Linux Kernel Documentation, 2020.

[18] G. Gallant. *The State of WebAssembly - 2022 and 2023*. 2023. URL: https://platform.uno/blog/the-state-of-webassembly-2022-and-2023/ (visited on 08/13/2023).

[19] Google. *V8 JavaScript and WebAssembly Engine*. URL: https://v8.dev/ (visited on 08/10/2023).

[20] 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*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 185–200. ISBN: 9781450349888. DOI: 10.1145/3062341.3062363.

[21] "IEEE Standard for Floating-Point Arithmetic." In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[22] *ISO/IEC 9899:2011 Information technology – Programming languages – C*. Tech. rep. Geneva, CH, 2011.

[23] W. Kluge. *Abstract Computing Machines*. Berlin, Heidelberg: Springer-Verlag, 2004. ISBN: 3540211462.

[24] D. K. Kumar. *The ABCs of Linux Signals: SIGINT, SIGTERM, and SIGKILL explained*. 2023. URL: https://www.fosslinux.com/121761/the-abcs-of-linux-signals-sigint-sigterm-and-sigkill-explained.htm (visited on 08/04/2023).

[25] H. Liljestrand, C. Chinea, R. Denis-Courmont, J.-E. Ekberg, and N. Asokan. *Color My World: Deterministic Tagging for Memory Safety*. 2022. arXiv: 2204.03781 [cs.CR].

[26] Linux. *siginfo.h*. 2023. URL: https://github.com/torvalds/linux/blob/master/include/uapi/asm-generic/siginfo.h (visited on 08/04/2023).

[27] LLVM. *The LLVM Compiler Infrastructure*. URL: https://llvm.org/ (visited on 08/04/2023).

[28] *malloc(3) - Linux manual page*. man7.org.

[29] C. Martínez, M. E. Nebel, and S. Wild. "Analysis of Branch Misses in Quicksort." In: *2015 Proceedings of the Twelfth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*. Society for Industrial and Applied Mathematics, Dec. 2014. DOI: 10.1137/1.9781611973761.11.

[30] A. E. Michael, A. Gollamudi, J. Bosamiya, C. Disselkoen, A. Denlinger, C. Watt, B. Parno, M. Patrignani, M. Vassena, and D. Stefan. *MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code*. 2022. arXiv: 2208.13583 [cs.CR].

[31] Microsoft. *Virtual Address Space*. URL: https://learn.microsoft.com/en-us/windows/win32/memory/virtual-address-space?redirectedfrom=MSDN (visited on 08/14/2023).

[32] Microsoft Security Response Center. *Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape*. 2019.

[33] Mozilla. *SpiderMonkey JavaScript and WebAssembly Engine*. URL: https://spidermonkey.dev/ (visited on 08/10/2023).

[34] Mozilla Developer Network. *Understanding the WebAssembly text format*. 2023. URL: https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format (visited on 08/04/2023).

[35] Mozilla Developer Network. *Using the WebAssembly JavaScript API*. 2023. URL: https://developer.mozilla.org/en-US/docs/WebAssembly/Using_the_JavaScript_API (visited on 08/04/2023).

[36] NSA Media Relations. *Software Memory Safety*. 2022.

[37] D. Page. *A Practical Introduction to Computer Architecture*. 1st. Springer Publishing Company, Incorporated, 2009. ISBN: 1848822553.

[38] QEMU Project. *QEMU: A generic and open source machine emulator and virtualizer*. URL: https://www.qemu.org/ (visited on 08/04/2023).

[39] Qualcomm Technologies, Inc. *Pointer Authentication on ARMv8.3: Design and Analysis of the New Software Security Instructions*. Tech. rep. Qualcomm Technologies, Inc., 2017.

[40] A. Rossberg. *GC Proposal for WebAssembly*. 2023. URL: https://github.com/WebAssembly/gc (visited on 08/13/2023).

[41] M. Rutland. *Pointer authentication in AArch64 Linux*. 2017. URL: https://docs.kernel.org/5.10/arm64/pointer-authentication.html (visited on 08/04/2023).

[42] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov. *Memory Tagging and how it improves C/C++ memory safety*. 2018. arXiv: 1802.09517 [cs.CR].

[43] Y. Shi, K. Casey, M. A. Ertl, and D. Gregg. "Virtual Machine Showdown: Stack versus Registers." In: *ACM Trans. Archit. Code Optim.* 4.4 (2008). ISSN: 1544-3566. DOI: 10.1145/1328195.1328197.

[44] *sigaction(2) — Linux manual page*. man7.org.

[45] W. Stallings. "Reduced instruction set computer architecture." In: *Proceedings of the IEEE* 76.1 (1988), pp. 38–55. DOI: 10.1109/5.3287.

[46] A. Taylor, A. Whalley, D. Jansens, and N. Oskov. *An update on Memory Safety in Chrome*. 2021.

[47] WebAssembly. *WASI libc implementation for WebAssembly*. URL: https://github.com/WebAssembly/wasi-libc (visited on 08/04/2023).

[48] WebAssembly. *WebAssembly*. URL: https://webassembly.org/ (visited on 08/04/2023).

[49] WebAssembly. *WebAssembly Documentation: Binary Encoding*. URL: https://webassembly.org/docs/binary-encoding/ (visited on 08/04/2023).

[50] WebAssembly. *WebAssembly Documentation: Textual Format*. URL: https://webassembly.org/docs/text-format/ (visited on 08/04/2023).

[51] WebAssembly. *WebAssembly Memory64 Proposal*. URL: https://github.com/WebAssembly/memory64/blob/main/proposals/memory64/Overview.md (visited on 08/10/2023).