MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code

ALEXANDRA E. MICHAEL*, UCSD, USA
ANITHA GOLLAMUDI*, University of Massachusetts, Lowell, USA
JAY BOSAMIYA, CMU, USA
EVAN JOHNSON, UCSD, USA
AIDAN DENLINGER, UCSD, USA
CRAIG DISSELKOEN, UCSD, USA
CONRAD WATT, University of Cambridge, UK
BRYAN PARNO, CMU, USA
MARCO PATRIGNANI, University of Trento, Italy
MARCO VASSENA, Utrecht University, Netherlands
DEIAN STEFAN, UCSD, USA

Most programs compiled to WebAssembly (Wasm) today are written in unsafe languages like C and C++. Unfortunately, memory-unsafe C code remains unsafe when compiled to Wasm—and attackers can exploit buffer overflows and use-after-frees in Wasm almost as easily as they can on native platforms. Memory-Safe WebAssembly (MSWasm) proposes to extend Wasm with language-level memory-safety abstractions to precisely address this problem. In this paper, we build on the original MSWasm position paper to realize this vision. We give a precise and formal semantics of MSWasm, and prove that well-typed MSWasm programs are, by construction, robustly memory safe. To this end, we develop a novel, language-independent memory-safety property based on *colored* memory locations and pointers. This property also lets us reason about the security guarantees of a formal C-to-MSWasm compiler—and prove that it always produces memory-safe programs (and preserves the semantics of safe programs). We use these formal results to then guide several implementations: Two compilers of MSWasm to native code, and a C-to-MSWasm compiler (that extends Clang). Our MSWasm compilers support different enforcement mechanisms, allowing developers to make security-performance trade-offs according to their needs. Our evaluation shows that on the PolyBenchC suite, the overhead of enforcing memory safety in software ranges from 22% (enforcing spatial safety alone) to 198% (enforcing full memory safety), and 51.7% % when using hardware memory capabilities for spatial safety and pointer integrity.

More importantly, MSWasm's design makes it easy to swap between enforcement mechanisms; as fast (especially hardware-based) enforcement techniques become available, MSWasm will be able to take advantage of these advances almost for free.

In the following, we use syntax highlighting accessible to both colourblind and black & white readers [Patrignani 2020]. Specifically, we use a blue, sans-serif *font for C and a* **red**, **bold** *font for MSWasm.*

1 INTRODUCTION

WebAssembly (Wasm) is a new bytecode designed to run native applications—e.g., applications written in C/C++ and Rust—at native speeds, everywhere—from the Web, to edge clouds, and IoT

*These authors contributed equally to this work

Authors' addresses: Alexandra E. Michael*, UCSD, USA; Anitha Gollamudi*, University of Massachusetts, Lowell, USA; Jay Bosamiya, CMU, USA; Evan Johnson, UCSD, USA; Aidan Denlinger, UCSD, USA; Craig Disselkoen, UCSD, USA; Conrad Watt, University of Cambridge, UK; Bryan Parno, CMU, USA; Marco Patrignani, University of Trento, Italy; Marco Vassena, Utrecht University, Netherlands; Deian Stefan, UCSD, USA;.

2022. XXXX-XXXX/2022/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

platforms. Unlike most industrial bytecode and compiler intermediate representations, Wasm was designed with safety in mind: Wasm programs run in an isolated sandbox *by construction*. On the Web, this means that Wasm programs cannot read or corrupt the browser's memory [Haas et al. 2017a]. On edge clouds, where Wasm programs written by different clients run in a single process, this means that one client cannot interfere with another [McMullen 2020].

Within the sandbox, however, Wasm offers little protection. Programs written in unsafe languages—and two thirds of existing Wasm programs are compiled from C/C++ [Hilbig et al. 2021]—remain unsafe when compiled to Wasm [Lehmann et al. 2020]. Indeed, buffer overflows and use-after-free vulnerabilities are as easy to exploit in Wasm as they are natively; sometimes even *easier* (e.g., because Wasm lacks abstractions like read-only memory). Worse, attackers can use such exploits to confuse the code hosting Wasm into perfoming unsafe actions—to effectively bypass the Wasm sandbox. [Lehmann et al. 2020], for example, show how attackers can turn a buffer overflow vulnerability in the `libpng` image processing library (executing in a Wasm sandbox) into a cross-site scripting (XSS) attack.

To prevent such attacks, C/C++ compilers would have to insert memory-safety checks *before* compiling to Wasm—e.g., to ensure that pointers are valid, within bounds, and point to memory that has not been freed [Nagarakatte et al. 2009, 2010; Necula et al. 2005]. Industrial compilers like Emscripten and Clang do not. Also, they *should not*. Retrofitting programs to enforce memory safety gives up on *robustness*, i.e., preserving memory safety when linking a (retrofitted) memory-safe module with a potentially memory-unsafe module. It gives up on *performance*: efficient memory-safety enforcement techniques rely on operating system abstractions (e.g., virtual memory [Dang et al. 2017]), abuse platform-specific details (e.g., encoding bounds information in the (unused) upper bits of an address [Akritidis et al. 2009]), and take advantage of hardware extensions (e.g., Arm's pointer authentication and memory tagging extensions [Arm 2019; Liljestrand et al. 2019]). Finally, it also makes it harder to prove that memory safety is preserved end-to-end.

With *Memory-Safe WebAssembly*, [Disselkoen et al. 2019] propose to bridge this gap by extending Wasm with language-level memory-safety abstractions. In particular, MSWasm extends Wasm with *segments*, i.e., linear regions of memory that can only be accessed using *handles*. Handles, like CHERI capabilities [Watson et al. 2015], are unforgeable, well-typed pointers—they encapsulate information that make it possible for MSWasm compilers to ensure that each memory access is valid and within the segment bounds. Alas, the MSWasm position paper only outlines this design—they do not give a precise semantics for MSWasm, nor implement or evaluate MSWasm as a memory-safe intermediate representation.

This paper builds on this work to realize the vision of MSWasm. We do this via five contributions:

1. Semantics and Memory Safety for MSWasm (Section 3). Our first contribution is a formal specification of MSWasm as an extension of the Wasm language, type system, and operational semantics. Our semantics give precise meaning to the previous informal design [Disselkoen et al. 2019]. Moreover, these semantics allow us to prove that all well-typed MSWasm programs are *robustly memory safe*; i.e., MSWasm programs are memory safe when linked against arbitrary code.

2. Color-based Memory-Safety Monitor (Section 4). We develop a novel, abstract memory-safety monitor based on *colored* memory locations and pointers, which we use to show that MSWasm is memory safe. Colors abstract away specific mechanisms that MSWasm backends can employ to enforce memory safety. Additionally, they enable reasoning about spatial as well as temporal memory safety, both at the granularity of individual memory objects and within structured objects as well. Furthermore, since our memory-safety monitor is language-independent, we can reason about memory-safety across compilation and establish the soundness of our compiler-based memory-safety enforcement in our next contribution.

3. Sound Compilation from C to MSWasm (Section 5). Like Wasm, MSWasm is intended to be used as a compilation target from higher-level languages. Hence, our third contribution is a formal C-to-MSWasm compiler, which guarantees memory-safe execution of unsafe code. In particular, we formalize a compiler from a subset of C to MSWasm and prove that the compiler soundly *enforces* memory-safety. Intuitively, this result ensures that memory-safe C programs when compiled to MSWasm remain safe and preserve their semantics, while memory-unsafe C programs trap at the first memory violation (and are thus safe too).

4. Implementations of MSWasm (Section 6). Our next contribution is the implementation of three MSWasm-related compilers. First, we implement an ahead-of-time (AOT) MSWasm-to-machine code compiler by extending the rWasm [Bosamiya et al. 2022] compiler with 1900 lines of code (LOC). Our extension of rWasm supports multiple options for enforcing memory safety, with tradeoffs between performance and differing levels of memory safety (spatial and temporal safety, and handle integrity). Our second compiler is a just-in-time (JIT) MSWasm-to-JVM compiler (1200 LOC), which uses the GraalVM Truffle framework [Oracle 2021b]. Finally, our third compiler is an LLVM-to-MSWasm compiler (1600 LOC) created as an extension of the CHERI Clang compiler toolchain [CTSRD-CHERI 2022].

5. Evaluation of MSWasm (Section 7). Our final contribution is an empirical evaluation of MSWasm. We benchmark MSWasm on PolyBenchC, the de-facto Wasm benchmarking suite [Pouchet 2011]. We find that, on (geomean) average, MSWasm when enforced in software using our AOT compiler imposes an overhead of 197.5%, which is comparable with prior work on enforcing memory safety for C [Nagarakatte et al. 2010]. MSWasm, however, makes it easy to change the underlying enforcement mechanism (e.g., to boost performance), without changing the application. To this end, we find that enforcing just spatial and temporal safety imposes a 52.2% overhead, and enforcing spatial safety alone using a technique similar to Baggy Bounds [Akritidis et al. 2009], is even cheaper—21.4%. Our JIT compiler, which enforces spatial and temporal safety, but not handle integrity, has an overhead of 42.3%. While these overheads are relatively large on today's hardware, upcoming hardware features explicitly designed for memory-safety enforcement can reduce these overheads (e.g., Arm's PAC can be used to reduce pointer integrity enforcement to under 20% [Liljestrand et al. 2019], while Arm's CHERI [Grisenthwaite 2019] or Intel's CCC [LeMay et al. 2021] can also reduce the cost of enforcing temporal and spatial safety). MSWasm will be able to take advantage of these features as soon they become available, as illustrated by the ease of swapping memory-safety enforcement techniques within our AOT compiler.

Open Source & Technical Report. Our technical report, implementations, benchmarks, and data sets are available as supplementary material and will be made open source.

2 BACKGROUND AND MOTIVATION

We now give a brief introduction to Wasm (Section 2.1), its attacker model (Section 2.2), and the implications of memory unsafety within the Wasm sandbox (Section 2.3). Then we give a brief introduction to MSWasm and to the open challenges we address in this work (Section 2.4).

2.1 WebAssembly

Wasm is a low-level bytecode, designed as a safe compilation target for higher-level languages like C/C++ and Rust [Haas et al. 2017b]. Wasm bytecode is executed in a sandboxed environment by a stack-based virtual machine. Prior to execution, the virtual machine type-checks the bytecode to ensure that each instruction finds the appropriate operands on the stack. Wasm's type system is extremely simple; the language has four primitive types—32- and 64-bit integers and floats (i32 and i64, and f32 and f64 respectively)—and only structured control flow constructs (i.e., no gotos) which simplify type checking. The Wasm heap (or *linear memory*), however, is an untyped contiguous linear array of bytes. Instructions τ.load and τ.store allow values of the four primitive types to be read from and written to the memory at arbitrary integer offsets. At runtime, Wasm ensures that these accesses are in bounds (and *traps* when they are not).

This simple design makes whole classes of attacks impossible by design. For example, the type-system ensures that well-typed bytecode cannot hijack the virtual machine's control flow via stack-smashing attacks [One 1996]. The coarse-grained bounds-checks on memory accesses, together with structured control flow, confine Wasm to a sandbox [Tan 2017; Wahbe et al. 1993]—and thus prevent Wasm from harming its host environment.

This simple design has a trade-off: We necessarily lose information when compiling programs written in high-level languages to Wasm. Clang, for example, compiles complex source-level values (e.g., structs and arrays) into "bags of bytes" in the untyped linear memory and compiles pointers to Wasm 32-bit integers, offsets in the linear memory where values are layed out. This, unfortunately, means that misusing C/C++ pointers is as simple and severe in Wasm as it is for native platforms.

2.2 Threat Model

In this work, we consider a Wasm-level attacker who attempts to exploit a memory vulnerability present in a C program compiled to Wasm. We consider vulnerabilities that can be triggered by *spatial* memory errors (e.g., buffer overflows), *temporal* memory errors (e.g., use-after-free and double-free vulnerabilities), and *pointer integrity* violations (e.g., corrupting function pointers to bend control flow). We assume the vulnerable program is linked with arbitrary code written by the attacker, which can interact with the program in any way allowed by Wasm semantics. To exploit a vulnerability, the attacker code can supply malicious inputs to the program and abuse values (including pointers) returned by or passed to the program. We leave memory unsafety of C++ programs and type confusion vulnerabilities [Haller et al. 2016] for future work.

2.3 Sandboxing Without Memory Safety

Memory unsafe C programs, when compiled to Wasm, largely remain unsafe: They can run uninterrupted as long as their reads and writes stay within the bounds of the entire linear memory. Unfortunately, Wasm also lacks most mitigations we rely on today to deal with memory unsafety (e.g., memory protection bits and ASLR), so a program compiled to run within Wasm's sandbox may be more vulnerable than if it were running on bare metal [Lehmann et al. 2020].

```
char *trim_token( char *token ){
 char *trimmed = malloc( 1024 *sizeof(char) );
 int i = 0, j = 0;
 while (token[i++] == '␣');
 char next = token[--i];
 while (next ≠ '\0') {
  trimmed[j++] = next; // Possible buffer overflow
  next = token[++i];
 }
 trimmed[i] = '\0';
 return trimmed;
}
```

Listing 1. Vulnerable code adapted from libpng 1.6.37

To understand how source-level memory vulnerabilities persist across compilation, consider the C code snippet in Listing 1 from `libpng 1.6.37`. Function `trim_token` takes a pointer to a null-terminated string as input and returns a pointer to a dynamically-allocated copy of the string, trimmed of the leading whitespace characters. The first loop (Line 4) simply scans the string `token` and skips all the whitespace characters, while the second loop (Line 6–Line 9) copies the rest of the string into `trimmed` one character at the time, until it finds the null terminator. The vulnerability is on Line 7: the length of the string `token` after trimming may exceed the size allocated for buffer `trimmed`. To exploit this vulnerability, an attacker only needs to call this function on a sufficiently long string (longer than 1024 characters after trimming). This will cause the function to write past the bounds of `trimmed`, thus corrupting the memory of the program with the payload supplied by the attacker. This vulnerability remains in the code obtained by compiling function `trim_token` with existing Wasm compilers (e.g., Emscripten and Clang). In particular, Line 7 gets translated into the Wasm instructions in Listing 2.

```
get $trimmed
get $j
i32.add
get $next
i32.store
...   ;; increment $j
```

Listing 2. "Compilation of Line 7 into Wasm."

The first three Wasm instructions compute the address (a 32-bit integer) where the next character gets copied, by adding index **$j** to address **$trimmed**. Then, instruction **get $next** pushes the value of the next character on the stack and **i32.store** writes it to the address computed before. As long as this address is within the linear memory region, the store instruction succeeds—even if the address does not belong to the buffer allocated for **$trimmed**.

Although an attacker could not use this memory-safety vulnerability to escape Wasm's sandbox, they could use it to corrupt and steal data (e.g., private keys) sensitive to the Wasm program itself. Wasm programs on the Web already handle sensitive data, and as Wasm's adoption expands beyond the Web, addressing memory safety within the sandbox is crucial.

2.4 The MSWasm Proposal

Memory-Safe WebAssembly (MSWasm) addresses these challenges by extending Wasm with abstractions for enforcing memory safety [Disselkoen et al. 2019]. Specifically, MSWasm introduces a new memory region called *segment memory*. The segment memory consists of individual *segments*, which are linearly addressable, bounded regions of memory representing dynamic memory allocations. Unlike Wasm's linear memory, the segment memory cannot be accessed at arbitrary offsets through standard **load** and **store** instructions. Instead, MSWasm provides new types, values, and instructions to regulate access to segments and enforce per-allocation memory safety. Segments can only be accessed through *handles*, unforgeable memory capabilities that model pointers bounded to a particular allocation of the segment memory. MSWasm adopts this low-level memory model since an object-based model (like that of the JVM) would be an inefficient (due to garbage collection overhead) and overly restrictive (due to the constraints of an object-based type-system) compilation target for C code deployed to Wasm.

Handles are tuples: $\langle$**base**, **offset**, **bound**, **isCorrupted**, **id**$\rangle$, where **base** represents the beginning of the segment in segment memory, **offset** is the handle's offset *within* the segment, i.e., within the **bound**, that the handle points to. Thus, a handle points to the address given by **base + offset**. MSWasm guarantees *handle integrity* using the **isCorrupted** flag. Intuitively, attempts to forge handles (e.g., by casting an integer, or altering the bitstring representation of an existing handle in memory) result in a corrupted handle. MSWasm traps only when an out-of-bounds or corrupt handle is *used*, not when it is created. This improves both performance, by eliminating checks on every pointer-arithmetic operation, and compatibility, since many C idioms create benign out-of-bound pointers [Memarian et al. 2019a, 2016; Ruef et al. 2019]. Finally, MSWasm associates each segment allocation with a unique identifier **id**, which is used to enforce *temporal* memory safety.

MSWasm provides new instructions to create and manipulate handles, and to access segments safely through them. Instructions τ**.segload** and τ**.segstore** are analogous to τ**.load** and τ**.store**, but operate on handles and trap if the handle is corrupted or points outside the segment bounds, or if the segment has been freed. Instruction segalloc allocates a segment of the desired size in a free region of segment memory and returns a handle to it. Instruction segfree frees the segment associated with a valid handle, thus making that region of segment memory available for new allocations. Lastly, instruction handle.add is for pointer arithmetic and modifies the handle offset, without changing the base or bound.

Eliminating unsafety by compiling C to MSWasm. With MSWasm we can eliminate potential memory vulnerabilities automatically, via compilation. For example, a C to MSWasm compiler would emit the instructions in Listing 3 for the code snippet from Listing 1. This code allocates a new 1024-byte segment and stores the handle for it in variable $trimmed. Then, the handle.add instruction increments the offset of $trimmed with index $j and instruction i32.segment_store writes $next in the segment. Since MSWasm instructions enforce memory safety, this code is safe to execute even with malicious inputs. In particular, if the offset of $trimmed is incremented past the bound of the handle, the store instruction simply traps, thus preventing the buffer overflow.

```
i32.const 1024
segalloc
set $trimmed
...
get $trimmed
get $j
handle.add
get $next
i32.segment_store
...
```

Listing 3. "Compilation of Listing 1 into Wasm."

Enforcing Intra-Object Memory Safety. Through the abstractions described above, MSWasm enforces *inter-object* memory safety, i.e., at the granularity of individual allocations. Unfortunately, this alone is insufficient to prevent memory-safety violations within composite data types (e.g., structs), in which a pointer to a field overflows (or overruns) an adjacent field.

```
struct User {char name[32], int id };
struct User *my_user = malloc(sizeof(struct User));
char *my_name = my_user→name;
...
```

Listing 4. Intra-object memory safety vulnerability.

Consider the code snippet in Listing 4, which defines a struct object containing a fixed-length string `name` and an integer user `id`. When compiled to MSWasm, this code allocates a single segment for the `User` structure; thus the handle corresponding to `my_name` and derived from `my_user` via pointer arithmetic can also access field `id` without trapping. Therefore, an attacker could exploit a memory vulnerability in the code that manipulates `my_name` to corrupt the user `id` and impersonate another user.

Hence, to enforce *intra-object* memory safety, MSWasm provides an additional instruction called slice. Instruction slice shrinks the portion of the segment that a handle can access by growing its base and reducing its bound field by a given offset. By emitting a slice instruction with appropriate offsets for expression `my_user->name`, a compiler can generate a sliced handle that includes only the `name` field. As a result, if the attacker later tries to overflow `my_name`, the safety checks of the sliced handle will detect a violation and trap the execution, thus preventing the program from corrupting the user `id`.

The missing pieces. The original MSWasm position paper [Disselkoen et al. 2019] only outlines the basic abstractions we describe above. The position paper does not give a formal (or even informal) semantics for the proposed language extensions. They do not describe compilation techniques—how one would compile C code to MSWasm or how MSWasm would be compiled to native code—nor an implementation (and thus evaluation) of MSWasm. In this paper we address these limitations

$$\text{Modules } \mathbf{M} ::= \{\mathbf{funcs}\ \Phi^*, \mathbf{imports}\ \rho^*, \mathbf{heap}\ \mathbf{n_H}, \mathbf{segment}\ \mathbf{n_S}\}$$

$$\text{Fun. Defs } \Phi ::= \{\mathbf{var}\ \tau^*, \mathbf{body}\ \mathbf{i}^*\} : \rho \quad \text{Instructions Types } \rho ::= \tau^* \to \tau^*$$

$$\text{Value Types } \tau ::= \mathbf{i32} \mid \mathbf{i64} \mid \mathbf{f32} \mid \mathbf{f64} \mid \mathbf{handle}$$

$$\text{Instructions } \mathbf{i} ::= \mathbf{nop} \mid \mathbf{trap} \mid \tau.\mathbf{const}\ \mathbf{c} \mid \tau.\otimes \mid \mathbf{get}\ \mathbf{n} \mid \mathbf{set}\ \mathbf{n} \mid \tau.\mathbf{load} \mid \tau.\mathbf{store} \mid \mathbf{branch}\ \mathbf{i}^*\ \mathbf{i}^* \mid \mathbf{call}\ \mathbf{n}$$
$$\mid \mathbf{return} \mid \tau.\mathbf{segload} \mid \tau.\mathbf{segstore} \mid \mathbf{slice} \mid \mathbf{segalloc} \mid \mathbf{handle.add} \mid \mathbf{segfree}$$

Fig. 1. Syntax of MSWasm with extensions to Wasm highlighted.

and, for the first time, provide an end-to-end, robust, memory-safe C-to-MSWasm compiler that is rooted in formal methods.

3 THE MSWASM LANGUAGE

This section develops a formal model of the design of MSWasm described above. The model includes syntax, typing (Section 3.1), and operational semantics for MSWasm (Section 3.2) and it serves as a specification for different low-level mechanisms (bounds checks, segment identifiers, integrity tags, etc.) needed to enforce memory safety in MSWasm. We present the properties of MSWasm in the next section, after formally defining memory safety.

Due to space constraints, we present a selection of the formalization, and elide proofs and auxiliary lemmas. The interested reader can find these omissions in the supplementary material.

3.1 MSWasm Syntax

The syntax of MSWasm is defined in Figure 1. MSWasm programs are modules $\mathbf{M}$, which specify a list of function definitions Φ^*, the type of imported functions ρ^*, and the size of the linear and segment memory ($\mathbf{n_H}$ and $\mathbf{n_S} \in \mathbb{N}$).[1] Syntax $\{\mathbf{var}\ \tau^*, \mathbf{body}\ \mathbf{i}^*\} : \rho$ defines a function with local variables of types τ^*, body $\mathbf{i}^*$, and function type ρ. Instructions $\mathbf{i}$ manage the operand stack and are mostly standard. Variables are referred to through numeric indices $\mathbf{n}$, which are statically validated during type-checking. For example, instructions $\mathbf{get}\ \mathbf{n}$ and $\mathbf{set}\ \mathbf{n}$ retrieve and update the value of the $\mathbf{n}$-th local variable, respectively. Function calls are similar, i.e., instruction $\mathbf{call}\ \mathbf{n}$ calls the $\mathbf{n}$-th function (either defined or imported) in the scope of the module. We describe MSWasm's instructions on segments and handles below.

Typing. The type system of MSWasm is a straightforward extension of Wasm's, and it similarly guarantees type safety (i.e., well-typed modules satisfy progress and preservation). Instructions are typed by the judgment $\Gamma \vdash \mathbf{i} : \tau_1^* \to \tau_2^*$, where τ_1^* and τ_2^* are the types of the values that $\mathbf{i}$ pops and pushes on the stack, respectively, and the typing context Γ tracks the type of the variables and functions in scope. Compared to Wasm, the only restriction imposed by the type system of MSWasm is that it prevents programs from forging handles by reading raw bytes from the unmanaged linear memory, i.e., $\Gamma \vdash \tau.\mathbf{load} : [i32] \to [\tau]$ iff $\tau \neq \mathbf{handle}$.

[1]We use e^* to denote a list of e elements, and e^n for a list of length n. We write $[e_0, e_1, \cdots]$ for finite lists, $[\,]$ for the empty list, $e : e^*$ to add e in front of e^*, and $e_1{}^* + e_2{}^*$ to append $e_2{}^*$ to $e_1{}^*$. Notation $e^*[i]$ looks up the i-th element of e^* and $e^*[i \mapsto v]$ replaces the i-th element of e^* with v.

$$\text{Store } \Sigma ::= (H, T, A) \qquad \text{Heaps } H ::= b^* \qquad \text{Bytes } b \in \{0..2^8\text{-}1\}$$

$$\text{Constants } c \qquad \text{Segments } T ::= (b, t)^* \qquad \text{Allocators } A$$

$$\text{Local Frames } F ::= (\theta, i^*, v^*) \qquad \text{Locals } \theta ::= (n \mapsto v)^* \qquad \text{Segment Tags } t ::= \bigcirc \mid \square$$

$$\text{Values } v ::= c \mid h \qquad \text{Handles } h ::= \langle n_{base}, n_{offset}, n_{bound}, b_{valid}, n_{id} \rangle$$

$$\text{Memory Events } \alpha ::= \epsilon \mid \text{read}_\tau(h) \mid \text{write}_\tau(h) \mid \text{salloc}(h) \mid \text{sfree}(h) \mid \text{trap}$$

Fig. 2. Wasm and MSWasm runtime structures.

(Stack-Top)
$$\frac{\Phi^* \vdash \langle \Sigma, (\theta, i, v^*) \rangle \xrightarrow{\alpha}\!\!\!\!\rightarrow \langle \Sigma', (\theta', i'^*, v'^*) \rangle}{\Phi^* \vdash \langle \Sigma, (\theta, i : i^*, v^* +\!\!+ v_b{}^*) \rangle \xrightarrow{\alpha}\!\!\!\!\rightarrow \langle \Sigma, (\theta, i'^* +\!\!+ i^*, v'^* +\!\!+ v_b{}^*) \rangle}$$

(Get)
$$\frac{\theta[n] = v}{\Phi^* \vdash \langle \Sigma, (\theta, \text{get } n, []) \rangle \twoheadrightarrow \langle \Sigma, (\theta, [], [v]) \rangle}$$

(If-T)
$$\frac{n \neq 0}{\Phi^* \vdash \langle \Sigma, (\theta, \text{branch } i_1{}^* \; i_2{}^*, [n]) \rangle \twoheadrightarrow \langle \Sigma, (\theta, i_1{}^*, []) \rangle}$$

(Load)
$$\frac{0 \leq n \qquad n + |\tau| < |H| \qquad b^{|\tau|} = [\, \Sigma.H[n+j] \mid j \in \{0..|\tau|-1\} \,] \qquad v = \tau.\text{unpack}(b^{|\tau|})}{\Phi^* \vdash \langle \Sigma, (\theta, \tau.\text{load}, [n]) \rangle \twoheadrightarrow \langle \Sigma, (\theta, [], [v]) \rangle}$$

Fig. 3. Semantics of Wasm (excerpts).

3.2 MSWasm Operational Semantics

To reason about the memory-safety guarantees of MSWasm, we define a small-step labeled operational semantics, which generates events for memory-relevant operations such as segment allocations and accesses.

3.2.1 Semantics of Wasm. Figure 2 defines the runtime structures used in the semantics judgment. A local configuration $\langle \Sigma, F \rangle$ consists of the store Σ and the stack frame F of the function currently executing. In Wasm, the store Σ contains only the unmanaged linear memory H, which is a list of bytes b^* of fixed length. The local stack frame F maintains the environment θ for variable bindings (mapping from variable indices to values), a list of instructions i^* to be executed, and the operand stack v^* for the values produced (and consumed) by those instructions. Values include constants c and integers n.

The semantics judgment $\Phi^* \vdash \langle \Sigma, F \rangle \xrightarrow{\alpha}\!\!\!\!\rightarrow \langle \Sigma', F' \rangle$ indicates that under function definitions Φ^*, local configuration $\langle \Sigma, F \rangle$ executes a single instruction and steps to $\langle \Sigma', F' \rangle$, generating event α (explained below). The semantics features also a separate judgment for function calls and returns, which is standard and omitted.

Figure 3 presents a selection of rules that MSWasm inherits from Wasm. Auxiliary rule (Stack-Top) extracts the first instruction and its operands from the list of instructions and the stack, respectively, and executes the instruction using the rules for individual instructions. Rule (Get) executes instruction get n, which looks up the value of variable n in the environment θ, i.e.,

$\theta[n] = v$, and pushes v on the stack. Instruction $\mathbf{branch}\ i_1^*\ i_2^*$ pops the integer condition n from the stack and returns instructions i_1^* if n is non-zero via rule (If-T).[2] Rule (Load) loads a value of type τ from address n in linear memory. Since the linear memory consists of plain bytes, the rule reads $|\tau|$ bytes at address n into byte string $b^{|\tau|}$ and converts them into a value of type τ, i.e., $v = \tau.\mathbf{unpack}(b^{|\tau|})$, which is then pushed on the stack. In the rule, premises $0 \leq n$ and $n + |\tau| < |H|$ ensure that the load instruction does not read outside the bounds of the linear memory, but do not enforce memory safety, as explained above.

3.2.2 *Semantics of MSWasm.*

MSWasm extends the runtime structures of Wasm with a managed segment memory, handle values, and a memory allocator (see Figure 2). The segment memory T is a fixed-length list $(b, t)^*$ of *tagged* bytes, where each tag t indicates whether the corresponding byte is part of a numeric value ($t = \circ$) or a handle ($t = \square$). These tags are used to detect forged or corrupted handles stored in segment memory and thus ensure handle integrity.

Handles $\langle n_{base}, n_{offset}, n_{bound}, b_{valid}, n_{id} \rangle$ contain the base address n_{base} of the memory region they span, length n_{bound}, offset n_{offset} from the base, integrity flag b_{valid} which indicates whether the handle is authentic ($b_{valid} = \mathbf{true}$) or corrupted ($b_{valid} = \mathbf{false}$), and segment identifier n_{id}. Finally, MSWasm instructions generate memory events α, which include the silent event ϵ, reading and writing values of type τ through a handle h (i.e., $\mathbf{read}_\tau(h)$ and $\mathbf{write}_\tau(h)$), segment allocations $\mathbf{salloc}(h)$, segment free $\mathbf{sfree}(h)$, and $\mathbf{trap}$ which is raised in response to a memory violation.

Memory Allocator. The MSWasm runtime system is responsible for providing a memory allocator to serve memory allocations of compiled programs. In our model, we represent the state of the memory allocator and its semantics explicitly, as this simplifies reasoning about memory safety. The allocator state A keeps track of free and used regions of segment memory and their identifiers, i.e., $A.\mathbf{free}$ and $A.\mathbf{allocated}$, respectively. The allocator serves allocation requests via reductions of the form $\langle T, A \rangle \xhookrightarrow{\mathbf{salloc}(a,n,n_{id})} \langle T', A' \rangle$, which allocates and initializes a free segment of n bytes, which starts at address a in segment memory and can be identified by fresh identifier n_{id}. Dually, reductions of the form $\langle T, A \rangle \xhookrightarrow{\mathbf{sfree}(a,n_{id})} \langle T', A' \rangle$ free the segment identified by n_{id} and allocated at address a, or traps, if no such segment is currently allocated at that address. We omit further details about the allocator state and semantics—the memory-safety guarantees of MSWasm do not depend on the concrete allocation strategy.

MSWasm Rules. Figure 4 gives some important rules for the new instructions of MSWasm. Rule (H-Load) loads a non-handle value ($\tau \neq \mathbf{handle}$) from segment memory through a *valid* handle $\langle n_1, o, n_2, \mathbf{true}, n_{id} \rangle$. Rule (H-Load) reads bytes b^* from the address pointed to by the handle, i.e., $n = n_1 + o$, and converts them into a value of type τ, i.e., $v_2 = \tau.\mathbf{unpack}(b^*)$.[3] The rule enforces memory safety by checking that (1) the handle is *not* corrupted, (2) the load does not read bytes outside the bounds of the segment, i.e., $0 \leq o$ and $o + |\tau| < n_2$, and (3) the segment is still allocated, i.e., $n_{id} \in A.\mathbf{allocated}$. Rule (H-Load-Handle) is similar, but for loading values of type $\mathbf{handle}$; therefore it includes additional checks (highlighted in gray), to enforce handle integrity. First, the rule checks that all the bytes read from memory are tagged as handle bytes, i.e., $b_c = \bigwedge_{t \in t^*}(t = \square)$, and then combines this flag with the flag b'_c obtained from the raw bytes of the segment; i.e., it returns handle $\langle n'_1, a', n'_2, b_c \wedge b'_c, n_{id} \rangle$. The combined flag invalidates handles obtained from bytes tagged as data, thus preventing programs from forging handles by altering their byte representation

---

[2] Wasm does not provide instructions for *unstructured control-flow*, common on native architectures (e.g., `JMP` on x86). Wasm code can define and jump to typed *labeled* blocks. Since these features do not affect the memory safety guarantees of MSWasm, we omit them from our model.

[3] Total function $\tau.\mathbf{unpack}$ converts $|\tau|$ bytes (the number of bytes needed to represent a value of type τ) into a value of type τ. The inverse function $\tau.\mathbf{pack}$ converts values to their byte representation.

(H-Load)

$$\frac{\begin{array}{cccc}\tau \neq \mathrm{handle} & v_1 = \langle n_1, o, n_2, \mathrm{true}, n_{id}\rangle & 0 \leq o & (b^*, \_) = [\ \Sigma.T[n+j] \mid j \in \{0..|\tau|-1\}\ ] \\ o + |\tau| < n_2 & n = n_1 + o & n_{id} \in \Sigma.A.\mathrm{allocated} & v_2 = \tau.\mathrm{unpack}(b^*)\end{array}}{\Phi^* \vdash \langle \Sigma, (\theta, \tau.\mathrm{segload}, [v_1])\rangle \xrightarrow{\mathrm{read}_\tau(v_1)} \langle \Sigma, (\theta, [], [v_2])\rangle}$$

(H-Load-Handle)

$$\frac{\begin{array}{cccc}\tau = \mathrm{handle} & v_1 = \langle n_1, o, n_2, \mathrm{true}, n_{id}\rangle & 0 \leq o & (b^*, t^*) = [\ \Sigma.T[n+j] \mid j \in \{0..|\tau|-1\}\ ] \\ o + |\tau| < n_2 & n = n_1 + o & n_{id} \in \Sigma.A.\mathrm{allocated} & n \,\%\, |\mathrm{handle}| = 0 \\ b_c = \bigwedge_{t \in t^*} (t = \square) & \tau.\mathrm{unpack}(b^*) = \langle n_1', a', n_2', b_c', n_{id}'\rangle & & v_2 = \langle n_1', a', n_2', b_c \wedge b_c', n_{id}'\rangle\end{array}}{\Phi^* \vdash \langle \Sigma, (\theta, \tau.\mathrm{segload}, [v_1])\rangle \xrightarrow{\mathrm{read}_\tau(v_1)} \langle \Sigma, (\theta, [], [v_2])\rangle}$$

(H-Store)

$$\frac{\begin{array}{ccccc}\tau \neq \mathrm{handle} & v_1 = \langle n_1, o, n_2, \mathrm{true}, n_{id}\rangle & o \geq 0 & o + |\tau| < n_2 & n_{id} \in \Sigma.A.\mathrm{allocated} \\ b^* = \tau.\mathrm{pack}(v_2) & t = \bigcirc & a = (n_1 + o) & \multicolumn{2}{c}{\Sigma'.T = \Sigma.T[a + j \mapsto (b_j, t) \mid j \in \{0..|\tau|-1\}]}\end{array}}{\Phi^* \vdash \langle \Sigma, \theta, \tau.\mathrm{segstore}, [v_2, v_1]\rangle \xrightarrow{\mathrm{write}_\tau(v1,v2)} \langle \Sigma', \theta, [], []\rangle}$$

(H-Store-Handle)

$$\frac{\begin{array}{cccc}\tau = \mathrm{handle} & v_1 = \langle n_1, o, n_2, \mathrm{true}, n_{id}\rangle & o \geq 0 & o + |\tau| < n_2 \\ n_{id} \in \Sigma.A.\mathrm{allocated} & & b^* = \tau.\mathrm{pack}(v_2) & t = \square \\ a = n_1 + o & a\%|\mathrm{handle}| = 0 & \multicolumn{2}{c}{\Sigma'.T = \Sigma.T[a + j \mapsto (b_j, t) \mid j \in \{0..|\tau|-1\}]}\end{array}}{\Phi^* \vdash \langle \Sigma, \theta, \tau.\mathrm{segstore}, [v_2, v_1]\rangle \xrightarrow{\mathrm{write}_\tau(v1,v2)} \langle \Sigma', \theta, [], []\rangle}$$

(H-Alloc)

$$\frac{\Sigma = (H, T, A) \qquad \langle T, A\rangle \xhookrightarrow{\mathrm{salloc}(a,n,n_{id})} \langle T', A'\rangle \qquad v = \langle a, 0, n, \mathrm{true}, n_{id}\rangle \qquad \Sigma' = (H, T', A')}{\Phi^* \vdash \langle \Sigma, (\theta, \mathrm{segalloc}, [n])\rangle \xrightarrow{\mathrm{salloc}(v)} \langle \Sigma', (\theta, [], [v])\rangle}$$

(H-Free)

$$\frac{\Sigma = (H, T, A) \qquad h = \langle a, 0, \_, \mathrm{true}, n_{id}\rangle \qquad \langle T, A\rangle \xhookrightarrow{\mathrm{sfree}(a,n_{id})} \langle T', A'\rangle \qquad \Sigma' = (H, T', A')}{\Phi^* \vdash \langle \Sigma, \theta, \mathrm{segfree}, [h]\rangle \xrightarrow{\mathrm{sfree}(h)} \langle \Sigma', \theta, [], []\rangle}$$

(Handle-Add)

$$\frac{v = \langle n_1, o, n_2, b, n_{id}\rangle \qquad v' = \langle n_1, o + n, n_2, b, n_{id}\rangle}{\Phi^* \vdash \langle \Sigma, (\theta, \mathrm{handle.add}, [n, v])\rangle \twoheadrightarrow \langle \Sigma, (\theta, [], [v'])\rangle}$$

(Slice)

$$\frac{v = \langle n_1, o, n_2, b, n_{id}\rangle \qquad 0 \leq o_1 < n_2 \qquad 0 \leq o_2 \qquad v' = \langle n_1 + o_1, o, n_2 - o_2, b, n_{id}\rangle}{\Phi^* \vdash \langle \Sigma, (\theta, \mathrm{slice}, [v, o_2, o_1])\rangle \twoheadrightarrow \langle \Sigma, (\theta, [], [v'])\rangle}$$

Fig. 4. Semantics of MSWasm (excerpts). The premises that ensure handle integrity are highlighted.

in memory. Furthermore, to enforce handle integrity, the rule allows loading handle values only from $|\mathrm{handle}|$-aligned memory addresses, i.e., $(n_1 + o)\%|\mathrm{handle}| = 0$. The alignment requirement is needed to avoid crafting fake handles. In fact, if one were to store two handles next to each other and then load from an address *within* the first one, the load would succeed and load bytes that all have the capability tag. However, the loaded value would be a fake capability, since the loaded bytes would be part of the first capability, and part of the second. Loading and storing at aligned addresses prevents this issue. The rules for $\tau.\mathrm{segstore}$ are analogous—they include similar bounds checks and alignment restrictions for handles—and additionally set the tag of the bytes that they write in memory according to τ. For example, Rule (H-Store) applies to values whose type are not handle, therefore it tags the bytes of the value written to memory as data (○). In contrast, Rule (H-Store-Handle) writes a **handle** to memory and so it tags its bytes accordingly (i.e., □).

Rule (H-Alloc) invokes the allocator to allocate and initialize a new segment of $\mathbf{n}$ bytes at address $\mathbf{a}$ in segment memory, and returns a handle to it. Rule (H-Free) invokes the allocator to free the segment bound to the given *valid* handle. Rule (Handle-Add) increments the offset of a handle $\mathbf{v}$, without changing the other fields. Notice that this rule allows programs to create handles that point out of bounds; out-of-bounds handles only cause a trap when they are used to access memory. Rule (Slice) creates a sliced handle $\langle \mathbf{n_1} + \mathbf{o_1}, \mathbf{o}, \mathbf{n_2} - \mathbf{o_2}, \mathbf{b}, \mathbf{n_{id}} \rangle$, where the base is increased by offset $\mathbf{o_1}$ and the bound is reduced by offset $\mathbf{o_2}$. Premises $\mathbf{0} \leq \mathbf{o_1} < \mathbf{n_2}$ and $\mathbf{0} \leq \mathbf{o_2}$ ensure that the handle obtained after slicing can only access a subset of the segment accessible from the original handle.

Whenever a τ**.segload**, a τ**.segstore**, a **segfree**, or a **slice** do not match their premise, the semantics traps, emitting a **trap** action and halting the execution immediately, with no values on the operand stack (omitted for brevity).

4 ABSTRACT MEMORY-SAFETY MONITOR

This section presents an abstract notion of memory safety that is based on *colored* memory locations (Section 4.1). Colors soundly abstract away many implementation details, which in turn let us formalize memory safety compactly as a trace property checked by a corresponding monitor (Section 4.2). We use this monitor to establish the spatial and temporal memory-safety guarantees of MSWasm (Section 4.3). Since our monitor is language-independent, we will reuse it to prove our C-to-MSWasm secure compiler enforces memory safety (Section 5).

4.1 Color-based Memory Safety

Our notion of memory safety associates pointers and memory locations with *colors* (which represent pointer provenance [Memarian et al. 2019b]), shades, and allocation tags. Intuitively, each memory allocation generates a pointer annotated with a *unique* color (and shades as described below) and assigns the same color to each location in the allocated region of memory. Then, we consider a memory access *spatially* safe if the color of the pointer corresponds to the color of the memory location it points to. To account for *temporal* safety, memory locations are tagged as free or allocated and we enforce that accessed locations are tagged as allocated.

Colors are suitable to reason about memory safety at the granularity of individual memory objects. In particular, this simple model is sufficient for low-level languages that do not natively support composite data types (e.g., Wasm and MSWasm). However, colors alone cannot capture *intra-object* memory violations (e.g., the vulnerability in Listing 4). Intuitively, this is because the simple model assigns the *same* color to all the fields of a struct object. To reason about intra-object safety, we thus extend colors with *shades* and use a different shade to decorate the memory locations of each field in a struct. As a result, a pointer to a struct field cannot be used to access another field of the same struct, as their shades do not match.

As explained above, our definition of memory safety is intentionally minimal and language-agnostic: it does not specify other operations on colored pointers, e.g., pointer arithmetic, and how they propagate colors. This lets us reuse this definition of memory safety for different languages and reason about enforcing memory-safety via compilation in Section 5.

4.2 Memory-Safety Monitor

We formalize our notion of memory safety by constructing a safety monitor [Schneider 2000], i.e., a state machine that checks whether a trace satisfies memory safety. Intuitively, the monitor consumes

$$\frac{T(a) = A(c,s)}{\alpha^* \vdash T \xrightarrow{\mathtt{read}(a^{(c,s)})} T}\ \text{(MS-Read)} \qquad \frac{T(a) = A(c,s)}{\alpha^* \vdash T \xrightarrow{\mathtt{write}(a^{(c,s)})} T}\ \text{(MS-Write)}$$

$$\frac{\mathtt{fresh}\,(c) \quad \forall j \in \{0..n-1\}.\, T(a+j) = F(\_,\_) \quad T' = T[a+i \mapsto A(c,\phi(i)) \mid i \in \{0..n-1\}]}{\alpha^* \vdash T \xrightarrow{\mathtt{alloc}(n,a^c,\phi)} T'}\ \text{(MS-Alloc)}$$

$$\frac{\mathtt{sfree}\,(a^c) \notin {\alpha_2}^* \quad T' = T[i \mapsto F(c,s_i) \mid i \mapsto A(c,s_i) \in T]}{{\alpha_1}^* \cdot \mathtt{alloc}(n,a^c,\phi) \cdot {\alpha_2}^* \vdash T \xrightarrow{\mathtt{sfree}(a^c)} T'}\ \text{(MS-Free)}$$

Fig. 5. Trace-based definition of memory safety.

a trace of memory events and gets stuck when it encounters a memory violation. We assume an infinite set of colors C, shades $\mathcal{S}$, and define a *colored shadow memory* $T \in \mathbb{N} \rightharpoonup \{A(c,s), F(c,s)\}$, i.e., a finite partial map from addresses $a \in \mathbb{N}$ to tagged colors $c \in C$ and shades $s \in \mathcal{S}$, where tags A and F denote whether a memory location is allocated or free, respectively. Then, we define an *abstract trace model* of memory events α, which include read and write operations with colored pointers, i.e., $\mathtt{read}(a^{(c,s)})$ and $\mathtt{write}(a^{(c,s)})$, memory allocations, i.e., $\mathtt{alloc}(n, a^c, \phi)$ denoting a n-sized c-colored allocation starting at address a, in which sub-regions are shaded according to function $\phi : \{0, \ldots, n-1\} \rightarrow \mathcal{S}$, and free operations, i.e., $\mathtt{sfree}\,(a^c)$ which frees the c-colored memory region allocated at address a. Lastly, we define the transition system of the monitor over shadow memories and event history α^* through the judgment $\alpha^* \vdash T \xrightarrow{\alpha} T'$ (Fig. 5).

Rules (MS-Read) and (MS-Write) consume events $\mathtt{read}(a^{(c,s)})$ and $\mathtt{write}(a^{(c,s)})$, respectively, provided that the color and the shade are equal to those stored at location a in shadow memory and that location a is allocated, i.e., $T(a) = A(c,s)$. If the colors or the shades do not match, or the memory location is free, the state machine simply gets stuck, thus detecting a memory violation. To consume event $\mathtt{alloc}(n, a^c, \phi)$, rule (MS-Alloc) allocates n contiguous, currently *free* locations in shadow memory, starting at address a, and assigns fresh color c and the shade given by ϕ to them. In response to event $\mathtt{sfree}\,(a^c)$, the monitor frees the c-colored region of memory previously allocated at address a through rule (MS-Free). First, the rule checks that a matching allocation event is present in the history, i.e., ${\alpha_1}^* \cdot \mathtt{alloc}(n, a^c, \phi) \cdot {\alpha_2}^*$ for some size n and shading function ϕ, and that region has not already been freed, i.e., $\mathtt{sfree}\,(a^c) \notin {\alpha_2}^*$, and then sets the tag of the memory locations colored c as free.

We say a trace is memory safe, written $\mathsf{MS}\,(\alpha^*)$, if and only if the state machine does not get stuck while processing the trace starting from the empty shadow memory $\varnothing$ and empty history ϵ. In the definition below, we write $\xrightarrow{\alpha^*}$ for the reflexive transitive closure of $\xrightarrow{\alpha}$, which accumulates single events into a trace and records the event history.

Definition 1 (Memory Safety). $\mathsf{MS}\,(\alpha^*) \stackrel{\text{def}}{=} \exists T.\, \epsilon \vdash \varnothing \xrightarrow{\alpha^*} T$

4.3 Memory Safety of MSWasm.

In order to establish memory safety for MSWasm, we first need to map the trace model of MSWasm to the abstract trace model of Section 4.1. The main difference between the two is that the abstract model identifies safe memory accesses using colors and shades, while MSWasm relies on bounds

$$\frac{\mathrm{h} = \langle \mathrm{n_b}, \mathrm{n_o}, \_, \_, \mathrm{n_{id}} \rangle \quad \delta(\mathrm{n_b}, \mathrm{n_{id}}) = b^{(c,s)} \quad a = b + \mathrm{n_o} \quad n = |\tau|}{\mathrm{read}_\tau\ (\mathrm{h}) =_\delta \mathtt{read}(a^{(c,s)}) \ \cdots\ \mathtt{read}((a+n-1)^{(c,s)})} \text{ (Trace-Read)}$$

$$\frac{\mathrm{h} = \langle \mathrm{n_b}, \mathrm{n_o}, \_, \_, \mathrm{n_{id}} \rangle \quad \delta(\mathrm{n_b}, \mathrm{n_{id}}) = b^{(c,s)} \quad a = b + \mathrm{n_o} \quad n = |\tau|}{\mathrm{write}_\tau\ (\mathrm{h}) =_\delta \mathtt{write}(a^{(c,s)}) \ \cdots\ \mathtt{write}((a+n-1)^{(c,s)})} \text{ (Trace-Write)}$$

$$\frac{\mathrm{h} = \langle \mathrm{n_b}, 0, \mathrm{n_o}, \_, \mathrm{n_{id}} \rangle \quad n = \mathrm{n_o} \quad \forall i \in \{0..n-1\}.\ \delta(\mathrm{n_b} + i, \mathrm{n_{id}}) = (a+i)^{(c,\phi(i))}}{\mathrm{salloc}(\mathrm{h}) =_\delta \mathtt{alloc}(n, a^c, \phi)} \text{ (Trace-SAlloc)}$$

$$\frac{\mathrm{h} = \langle \mathrm{n_b}, \_, \mathrm{n}, \_, \mathrm{n_{id}} \rangle \quad \delta(\mathrm{n_b}, \mathrm{n_{id}}) = a^{(c,\_)}}{\mathrm{sfree}(\mathrm{h}, \mathrm{n_{id}}) =_\delta \mathtt{sfree}\ (a^c)} \text{ (Tr-Sfree)} \qquad \frac{}{\mathrm{trap} =_\delta \epsilon} \text{ (Trace-Trap)}$$

Fig. 6. Relation between MSWasm and abstract events (excerpts).

checks and segment identifiers. Furthermore, individual $\mathrm{read}_\tau(\mathrm{h})$ and $\mathrm{write}_\tau(\mathrm{h})$ events correspond to multiple memory accesses in the abstract trace model, as these operations read and write byte sequences in MSWasm. We reconcile these differences between the two trace models with the relation $\alpha =_\delta \alpha^*$ whose most relevant rules are defined in Fig. 6. The relation is parametrized by a partial bijection $\delta : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N} \times C \times \mathcal{S}$, which maps pairs $(\mathrm{a}, \mathrm{n_{id}})$, consisting of an allocated segment memory address a and a segment identifier $\mathrm{n_{id}}$, into corresponding shadow memory addresses $a^{(c,s)}$, decorated with colors and shades. Intuitively, we can construct a suitable bijection δ from the MSWasm allocator, which has information about what is allocated in segment memory.

Rules (Trace-Read) and (Trace-Write) relate single MSWasm $\mathrm{read}_\tau\ (\mathrm{h})$ and $\mathrm{write}_\tau\ (\mathrm{h})$ events to a sequence of $|\tau|$ contiguous abstract read and write events, respectively. The rules convert the handle base address and the segment identifier into the corresponding colored base address, i.e., $\delta(\mathrm{n_b}, \mathrm{n_{id}}) = b^{(c,s)}$, which is then incremented with the offset of the handle to obtain the first abstract location accessed, i.e., $a = b + \mathrm{n_o}$, similar to MSWasm semantics. Since these abstract events originate from the same handle, the rule labels their address with the same color c and shade s obtained from the base address of the handle to reflect their provenance. If we computed the colors for these addresses using the bijection δ, then they would automatically match the color stored in the shadow memory and memory safety would hold trivially. Instead, these addresses are tagged with the provenance color, and therefore proving memory safety (i.e., stepping using the rules of Figure 5) requires showing that this color matches the color found in shadow memory, which in turn requires reasoning about the integrity of the handle and the bounds checks performed by MSWasm. A final subtlety of these rules is that they seem to ignore the integrity flag of the handle. This is because in MSWasm, only authentic handles can generate read and write events—reading and writing memory via corrupted handles results in a trap event.

Rule (Trace-SAlloc) relates the allocation of a $\mathrm{n_o}$-byte segment in MSWasm to a corresponding abstract allocation of the same size, i.e., event $\mathtt{alloc}(n, a^c, \phi)$ where $\mathrm{n_o} = n$. In the rule, premise $\forall i \in \{0, \ldots, n-1\}.\ \delta(\mathrm{a} + i) = (\mathrm{n_b} + i, \mathrm{n_{id}})^{(c,\phi(i))}$ ensure that (i) all the abstract addresses share the same color c, and (ii) the bijection δ and the shading function ϕ agree on the shades used for the segment. In general, function ϕ can be a constant function, when we reason about memory safety for native MSWasm programs, e.g., to prove that MSWasm is memory safe in Theorem 1 below. Intuitively, MSWasm does not provide an explicit representation for structured data, therefore it is sufficient to assign the same shade to all locations of a segment to prove memory safety. When we use MSWasm as a compilation target however, segments can store also structured objects (e.g., a struct) in addition to flat objects (e.g., an array). In this scenario, we instantiate ϕ according to the source type of the object, which let us show that compiled C/C++ programs achieve intra-object memory safety later (Theorem 2).

To relate free events, rule (Tr-Sfree) requires the bijection δ to match the base n_b and identifier n_{id} of the segment pointed to by the handle to the colored address a^c freed by the monitor, i.e., $\delta(n_b, n_{id}) = a^{(c,\_)}$. Because identifiers and colors are never reused, freed segments and regions can be reused for other allocations, while keeping dangling handles and colored pointers related by the bijection. For example, if a segment is later allocated at address n_b, it will be associated with a *unique* identifier $n'_{id} \neq n_{id}$, which can be related to some shadow address a' and *fresh* color $c' \neq c$ through an *extended* bijection $\delta' \supseteq \delta$.[4]

Lastly, rule (Trace-Trap) relates event trap in MSWasm to the empty trace ϵ, since trap simply stops the program and thus cannot cause a memory safety violation.

We can now state memory safety for MSWasm traces in terms of memory safety of a δ-related abstract trace for the state machine defined above.

Definition 2 (Memory Safety for MSWasm Traces). $\mathtt{MS}(\boldsymbol{\alpha}^*) \stackrel{\text{def}}{=} \exists \alpha^*, \boldsymbol{\delta}.\ \boldsymbol{\alpha}^* =_{\delta} \alpha^* \text{ and } \mathtt{MS}\,(\alpha^*)$

We define memory safety for MSWasm modules if the trace generated during execution is memory safe. In the following, we write $\mathbf{M} \rightharpoonup \boldsymbol{\alpha}^*$ for the trace generated by module $\mathbf{M}$ with the semantics of Section 3.2.

Definition 3 (Memory Safety for MSWasm Modules). $\vdash \mathtt{MS}(\mathbf{M}) \stackrel{\text{def}}{=} \mathbf{M} \rightharpoonup \boldsymbol{\alpha}^* \text{ and } \mathtt{MS}(\boldsymbol{\alpha}^*)$

A module $\mathbf{M}$ achieves robust memory safety if, given any valid attacker $\mathbf{C}$ (denoted as $\mathbf{M} \vdash \mathbf{C} : \mathbf{attacker}$, in the sense of Section 2.2), linking $\mathbf{M}$ with $\mathbf{C}$ produces a memory safe module. In the following, we write $\mathbf{M} \circ \mathbf{C}$ for the module obtained by linking $\mathbf{M}$ with $\mathbf{C}$, i.e., the module obtained by instantiating the functions imported by $\mathbf{M}$ with those of $\mathbf{C}$.

Definition 4 (Robust Memory Safety for MSWasm Modules). $\vdash \mathbf{RMS}\,(\mathbf{M}) \stackrel{\text{def}}{=} \forall \mathbf{C} \text{ s.t. } \mathbf{M} \vdash \mathbf{C} : \mathbf{attacker}. \vdash \mathtt{MS}(\mathbf{M} \circ \mathbf{C})$

The main result for MSWasm is that any well-typed module ($\vdash \mathbf{M} : \mathbf{wt}$) is memory-safe, robustly.

Theorem 1 (Robust Memory Safety for MSWasm). If $\vdash \mathbf{M} : \mathbf{wt}$ then $\vdash \mathbf{RMS}\,(\mathbf{M})$

Proof (Sketch). Intuitively, the type system of MSWasm ensures that well-typed modules can access segment memory only through handle values and safe instructions. Programs that accesses memory via *invalid* handles trap and so trivially respect memory safety (Rule (Trace-Trap)). When accessing segments via *valid* handles, MSWasm performs memory safety checks using their metadata, so the rest of the proof requires showing an invariant about handle integrity. Intuitively, this invariant guarantees that valid handles (whether proper values, or stored in segment memory), correspond to allocated segments in memory. Then, using this invariant, we can show that programs that access segment memory without trapping, pass the memory safety checks, and thus are memory safe. □

In the next section, we leverage the memory-safety abstractions of MSWasm to develop a formal C compiler that provably enforces memory safety.

5 MEMORY SAFETY THROUGH COMPILATION

This section shows how a C compiler targeting MSWasm can enforce memory safety. Thus, we formalize a simplified version of C (Section 5.1), as a memory-unsafe source language for our compiler, and the compiler itself (Section 5.2). We then prove that the compiler enforces memory safety

[4] In technical terms the bijection grows *monotonically*, which provides a suitable inductive principle for our formal results.

$$\text{Programs } M ::= I^*, D^*, F^*, n_{hs} \quad \text{Imports } I ::= \tau\ g(x : \tau) \quad \text{Structs } D ::= s \mapsto (f : \tau)^*$$
$$\text{Values } v ::= n \mid n^{(n,n,w,n_{id})} \qquad \text{Functions } F ::= \tau\ g(x : \tau)\ \{\ \mathsf{var}\ (y : \tau)^*, e\}$$
$$\text{Word Types } w ::= \tau \mid \mathsf{struct}\ s \mid \mathsf{array}\ \tau \quad \text{Expr. Types } \tau ::= \mathsf{int} \mid \mathsf{ptr}\ w$$
$$\text{Expr. } e ::= v \mid x \mid e; e \mid e \oplus e \mid x := e \mid x := \mathsf{call}\ g\ (e) \mid *e \mid e[e]$$
$$\mid *e := e \mid e[e] := e \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \&e \rightarrow f$$
$$\mid \mathsf{malloc}(\tau, e) \mid \mathsf{malloc}(w) \mid \mathsf{free}(e)$$
$$\text{Stores } \Sigma ::= \langle H, A\rangle \quad \text{Heaps } H ::= [\,] \mid v : H \quad \text{Local Env. } \theta ::= (x \mapsto v)^* \quad \text{Allocators } A$$
$$\text{Events } \alpha ::= \epsilon \mid \mathsf{alloc}(v) \mid \mathsf{free}(v) \mid \mathsf{read}_\tau(v) \mid \mathsf{write}_\tau(v)$$

Fig. 7. C syntax and runtime structures (excerpts).

(Section 5.3), i.e., memory-safe programs compiled to MSWasm execute unchanged (Theorem 2), while memory-unsafe programs abort at the first memory violation (Theorem 3).

5.1 The Source Language C

Figure 7 presents the syntax of our source language, a subset of C, inspired by previous work [Ruef et al. 2019]. Source programs M specify the type of imported functions I^*, struct definitions D^*, function definitions F^*, and the heap size n_{hs}. Struct definitions map struct names s to a list of field names f and their types τ. Types are mutually defined by expression types τ, i.e., integers (int) and pointers ($\mathsf{ptr}\ w$), and word types w, which, in addition to τ, include also multi-word values, i.e., structs ($\mathsf{struct}\ s$) and arrays ($\mathsf{ptr}\ \mathsf{array}\ \tau$). Note that arrays are always typed as pointers. Syntax $\tau\ g(x : \tau)\ \{\ \mathsf{var}\ (y : \tau)^*, e\}$ defines function g, its argument and return type, and declares local variables $(y : \tau)^*$ in scope of the body e. Expressions are standard and include reading and writing memory via pointers (i.e., $*e$ and $*e := e$) and accessing struct fields (i.e., $\&e \rightarrow f$). An array e_1 at index e_2 is read and written via $e_1[e_2]$ and $e_1[e_2] := e_3$.

Expression $\mathsf{malloc}(\tau, e)$ allocates an array containing e elements of type τ, while $\mathsf{malloc}(w)$ allocates a buffer to store a single element of type $w \neq \mathsf{array}\ \tau$. Values include integers n and annotated pointers, i.e., $n^{(n_1,n_2,w,n_{id})}$, where n is the address pointed to by the pointer, and (n_1, n_2, w, n_{id}) indicates that the pointer refers to a buffer allocated at address n_1, containing n_2 elements of type w, and identified by n_{id}. These annotations are inspired by previous work on pointer provenance [Memarian et al. 2019a] and are only needed to reason about memory safety of source programs, i.e., the source semantics does *not* enforce memory safety and ignores them.

Typing. The type system for the source language is mostly standard and defined by judgment $F^*, \Gamma \vdash e : \tau$, which indicates that expression e has type τ under functions F^* and typing context Γ (which binds variables to types). The type system allows typing integers as pointers and restricts function type signatures to expression types for simplicity.

Semantics. We define a small-step contextual semantics for C with the following judgment, $M \vdash \langle \Sigma, \theta, e\rangle \xrightarrow{\alpha} \langle \Sigma', \theta', e'\rangle$, in which local configuration $\langle \Sigma, \theta, e\rangle$ steps and produces event α, under program definition M. Local configurations contain the store Σ, the local variable environment θ mapping named variables to values, and an expression e to be evaluated. The store Σ contains the heap H, a list of values, and the allocator state A. The heap abstracts away low-level details about the memory layout and the byte representation of values (e.g., we store structs and arrays simply as a flattened sequence of single-word values). Similar to MSWasm, events α record memory relevant

(Ptr-Arith)

$$\frac{}{\mathsf{M} \vdash \langle \Sigma, \theta, \mathsf{a}^{(\mathsf{b},\ell,\mathsf{w},\mathsf{n_{id}})} \oplus \mathsf{n} \rangle \twoheadrightarrow \langle \Sigma, \theta, (\mathsf{a}+\mathsf{n})^{(\mathsf{b},\ell,\mathsf{w},\mathsf{n_{id}})} \rangle}$$

(Write-Ptr)

$$\frac{\Sigma = \langle \mathsf{H}, \mathsf{A} \rangle \quad \mathsf{H}' = \mathsf{H}[\mathsf{a} \mapsto \mathsf{v}] \quad \Sigma' = \langle \mathsf{H}', \mathsf{A} \rangle \quad \vdash \mathsf{v} : \tau}{\mathsf{M} \vdash \langle \Sigma, \theta, *\mathsf{a}^{(\mathsf{b},\ell,\mathsf{w},\mathsf{n_{id}})} := \mathsf{v} \rangle \xrightarrow{\mathsf{write}_\tau(\mathsf{a}^{(\mathsf{b},\ell,\mathsf{w},\mathsf{n_{id}})})} \langle \Sigma', \theta, 0 \rangle}$$

(Write-Int)

$$\frac{\Sigma = \langle \mathsf{H}, \mathsf{A} \rangle \quad \mathsf{H}' = \mathsf{H}[\mathsf{a} \mapsto \mathsf{v}] \quad \Sigma' = \langle \mathsf{H}', \mathsf{A} \rangle \quad \vdash \mathsf{v} : \tau}{\mathsf{M} \vdash \langle \Sigma, \theta, *\mathsf{a} := \mathsf{v} \rangle \xrightarrow{\mathsf{write}_\tau(\mathsf{a})} \langle \Sigma', \theta, 0 \rangle}$$

(Malloc-Single)

$$\frac{\Sigma = \langle \mathsf{H}, \mathsf{A} \rangle \xhookrightarrow{\mathsf{alloc}(\mathsf{a}^{(\mathsf{a},1,\mathsf{w},\mathsf{n_{id}})})} \langle \mathsf{H}', \mathsf{A}' \rangle = \Sigma' \quad \mathsf{v} = \mathsf{a}^{(\mathsf{a},1,\mathsf{w},\mathsf{n_{id}})}}{\mathsf{M} \vdash \langle \Sigma, \theta, \mathsf{malloc}(\mathsf{w}) \rangle \xrightarrow{\mathsf{alloc}(\mathsf{v})} \langle \Sigma', \theta, \mathsf{v} \rangle}$$

(Malloc-Array)

$$\frac{\Sigma = \langle \mathsf{H}, \mathsf{A} \rangle \xhookrightarrow{\mathsf{alloc}(\mathsf{a}^{(\mathsf{a},\mathsf{n},\tau,\mathsf{n_{id}})})} \langle \mathsf{H}', \mathsf{A}' \rangle = \Sigma' \quad \mathsf{v} = \mathsf{a}^{(\mathsf{a},\mathsf{n},\tau,\mathsf{n_{id}})}}{\mathsf{M} \vdash \langle \Sigma, \theta, \mathsf{malloc}(\tau, \mathsf{n}) \rangle \xrightarrow{\mathsf{alloc}(\mathsf{v})} \langle \Sigma', \theta, \mathsf{v} \rangle}$$

Fig. 8. Semantics of C (excerpts).

operations, including silent events ϵ, allocating and releasing memory, i.e., $\mathsf{alloc}(\mathsf{v})$ and $\mathsf{free}(\mathsf{v})$, and reading and writing values of type τ with a pointer v, i.e., $\mathsf{read}_\tau(\mathsf{v})$ and $\mathsf{write}_\tau(\mathsf{v})$.

Figure 8 presents some of the semantics rules of the source language. Rule (Ptr-Arith) performs pointer arithmetic by incrementing the address of the pointer, without changing the metadata. Rules (Write-Ptr) and (Write-Int) write a value v in the heap through a pointer and a raw integer address, respectively. As explained above, these rules do not check that the write operation is safe, but only record the pointer and the type τ of the value that gets stored in the generated event, i.e., $\mathsf{write}_\tau(\mathsf{a}^{(\mathsf{b},\ell,\mathsf{w},\mathsf{n_{id}})})$ and $\mathsf{write}_\tau(\mathsf{a})$. Rules (Malloc-Single) and (Malloc-Array) allocate a buffer for a single object of type w and a n-elements array, respectively, and return a pointer value annotated with appropriate metadata. Similar to the MSWasm semantics, the source language invokes the allocator to serve allocation and free requests ($\langle \mathsf{H}, \mathsf{A} \rangle \xhookrightarrow{\alpha} \langle \mathsf{H}', \mathsf{A}' \rangle$). In contrast to the safe allocator of MSWasm however, the source allocator does not trap upon an invalid free request, i.e., a free of an unallocated memory region, but silently drops the request.[5]

The source language uses a separate semantic judgment for function calls and returns (omitted), and a top-level judgment $\mathsf{M} \twoheadrightarrow \alpha^*$, which collects the trace generated by program M. We define memory safety for source traces using the general abstract monitor from Section 4.1:

Definition 5 (Memory-Safety for C). $\mathtt{MS}(\alpha^*) \overset{\text{def}}{=} \exists \alpha^*, \delta.\, \alpha^* =_\delta \alpha^*$ and $\mathsf{MS}\,(\alpha^*)$

This definition is analogous to Definition 2 for MSWasm: it relies on a bijection δ to map source addresses into corresponding colored abstract addresses, and a relation $\alpha^* =_\delta \alpha^*$ to connect source and abstract traces through the bijection. The trace relation is defined similarly to the relation given in Figure 6 for MSWasm and additionally constructs appropriate shading functions for $\mathsf{alloc}(\mathsf{v})$

[5] Invalid free requests cause *undefined behavior* in C and usually result in the corruption of memory objects or the allocator state. Since we represent the allocator state explicitly and separately from the program memory, free requests cannot cause such specific behaviors in our model.

(C-Ptr-Arith)
$$\frac{[\![P,\Gamma \vdash e_1 : \mathsf{array}\ \tau]\!]^{exp} = i_1^* \quad [\![P,\Gamma \vdash e_2 : \mathsf{int}]\!]^{exp} = i_2^* \quad \mathbf{n} = \mathsf{sz}\,(\tau)}{[\![P,\Gamma \vdash e_1 \oplus e_2 : \mathsf{array}\ \tau]\!]^{exp} = i_1^*; i_2^*; \mathbf{i32.const\ n; i32.\times; handle.add}}$$

(C-BinOp)
$$\frac{[\![P,\Gamma \vdash e_1 : w]\!]^{exp} = i_1^* \quad [\![P,\Gamma \vdash e_2 : w]\!]^{exp} = i_2^* \quad [\![\tau]\!] = \boldsymbol{\tau}}{[\![P,\Gamma \vdash e_1 \oplus e_2 : \tau]\!]^{exp} = i_1^*; i_2^*; \boldsymbol{\tau}.\otimes}$$

(C-Malloc-Array)
$$\frac{[\![P,\Gamma \vdash e : \mathsf{int}]\!]^{exp} = i_1^* \qquad \mathbf{n} = \mathsf{sz}\,(\tau)}{[\![\Gamma \vdash \mathsf{malloc}(\tau, e) : \mathsf{ptr}\,(\mathsf{array}\ \tau)]\!]^{exp} = {i_1}^*; \mathbf{i32.const\ n; i32.\otimes; segalloc}}$$

(C-Malloc-Single)
$$\frac{\mathbf{n} = \mathsf{sz}\,(w)}{[\![P,\Gamma \vdash \mathsf{malloc}(w) : \mathsf{ptr}\ w]\!]^{exp} = \mathbf{i32.const\ n; segalloc}}$$

(C-Deref)
$$\frac{[\![P,\Gamma \vdash e : \mathsf{ptr}\ \tau]\!]^{exp} = i^* \quad [\![\tau]\!] = \boldsymbol{\tau}}{[\![P,\Gamma \vdash *e : \tau]\!]^{exp} = i^* +\!\!+ [\boldsymbol{\tau}.\mathbf{segload}]}$$

(C-Struct-field)
$$\frac{[\![P,\Gamma \vdash e : \mathsf{ptr}\,(\mathsf{struct}\ s)]\!]^{exp} = i^* \quad (\mathbf{o}_1, \mathbf{o}_2) = \mathtt{offset}\,(s, f)}{[\![P,\Gamma \vdash \&e \to f : \mathsf{ptr}\ \tau]\!]^{exp} = i^* +\!\!+ [\mathbf{i32.const\ o_1, i32.const\ o_2, slice}]}$$

Fig. 9. Compiler from C to MSWasm (excerpts).

events according to the type of the allocated object (e.g., an array or a struct). Accesses via raw addresses n are excluded from the relation, i.e., $\mathsf{write}_\tau(a) \neq_\delta \alpha^*$ and $\mathsf{read}_\tau(a) \neq_\delta \alpha^*$ for any abstract trace α^*. Omitting them from the relation captures the fact that memory accesses with forged pointers violate memory safety, as the provenance of these pointers is undefined.

5.2 The Compiler

We define the compiler $[\![\cdot]\!]$ from C to MSWasm inductively on the type derivation of C modules, functions and expressions (Figure 9). To prevent untrusted code from violating memory safety, our compiler translates pointers to handles and only uses MSWasm segment memory. Thus, we translate source types τ into MSWasm types $\boldsymbol{\tau}$ as $[\![\mathsf{int}]\!] = \mathbf{i32}$ and $[\![\mathsf{ptr}\ w]\!] = \mathbf{handle}$. The compiler relies on source types to emit MSWasm instructions with appropriate byte sizes (calculated with function $\mathsf{sz}\,(\cdot) : \tau \to \mathbf{n}$) and offsets for expressions that involve pointer arithmetic, struct accesses and memory allocations. For example, a binary operation ($\oplus$) whose first operand is an array ($\mathsf{array}\ \tau$) needs to be compiled in a **handle.add**, as in Rule (C-Ptr-Arith). On the other hand, a binary operation on naturals needs to be compiled in the related MSWasm binary operation, as in Rule (C-BinOp). Another example of the way source types guide the compilation is for the compilation of expression $\mathsf{malloc}(\tau, e)$. Here, if the resulting type is a pointer to an array ($\mathsf{ptr}\,(\mathsf{array}\ \tau)$), the compiler must first emit instructions to compute the size of a segment large enough for an array containing e elements of type τ, and then instruction **segalloc** to invoke the allocator and generate the corresponding handle. Therefore, rule (C-Malloc-Array) recursively compiles the array length e, i.e., $[\![P,\Gamma \vdash e : \mathsf{int}]\!]^{exp} = i^*$, which then gets multiplied by $|\tau|$, i.e., the size in bytes of a value of type τ, via instruction $\mathbf{i32.\otimes}$, and finally passed to **segalloc**. On the other hand, if the return type is a pointer to any other type ($\mathsf{ptr}\ w$), the compiler needs to calculate its size ($\mathbf{n}$) and allocate enough memory (Rule (C-Malloc-Single)) Since expression $*e$ reads a pointer to a value of type τ, rule (C-Deref) emits instruction to first evaluate the corresponding handle, i.e., $[\![\mathsf{P}, \Gamma \vdash \mathsf{e} : \mathsf{ptr}\ \tau]\!]^{\mathsf{exp}} = \mathbf{i}^*$, followed by instruction $\boldsymbol{\tau}$**.segload**, whose compiled type $[\![\tau]\!] = \boldsymbol{\tau}$ ensures that the generated code reads the right number of bytes and interprets them at the corresponding target type. Lastly, rule (C-Struct-field) translates a struct field access $\&\mathsf{e} \longrightarrow \mathsf{f}$ by slicing the handle obtained from pointer e, thus enforcing intra-object safety in the generated code. To this end, the rule emits instructions $[\mathbf{i32.const}\ \mathbf{o}_1, \mathbf{i32.const}\ \mathbf{o}_2, \mathbf{slice}]$, where offsets **o1** and **o2** are obtained from function `offset` (s, f), which statically computes the offsets necessary to select field f in the byte representation of struct s.

5.3 Properties of the Compiler

We establish two properties for our compiler. The first (Theorem 2) shows that the compiler is functionally correct and preserves memory safety for memory-safe source programs. The second (Theorem 3) shows that *memory-unsafe* programs compiled to MSWasm abort at the first memory violation. Together, these results show that our compiler enforces memory-safety (Corollary 1).

Cross-Language Equivalence Relation. Since our notion of memory safety is defined over traces, and the source and target languages have different trace models, the formal results of the compiler rely on a *cross-language* equivalence relation to show functional correctness and memory-safety preservation [Leroy 2009]. Figure 10 (top) defines this relation for pointer values up to a partial bijection $\delta : \mathbb{N} \times \mathbb{N} \rightharpoonup \mathbb{N} \times \mathbb{N}$, which maps addresses and identifiers from source to target. Rule (Val-Rel-Ptr) relates an annotated pointer $\mathsf{a}^{(\mathsf{b},\ell,\mathsf{w},\mathsf{n_{id}})}$ to a *valid* handle $\langle \mathbf{b}, \mathbf{o}, \boldsymbol{\ell}, \mathbf{true}, \mathbf{n_{id}} \rangle$ as long as their base address and identifier are matched by the bijection, i.e., $\delta(\mathsf{b}, \mathsf{n_{id}}) = \mathbf{b}, \mathbf{n_{id}}$, and the length and offset fields match, taking into account the byte-size representation of w, i.e., $\ell \times |\mathsf{w}| = \boldsymbol{\ell}$ and $(\mathsf{a} - \mathsf{b}) \times |\mathsf{w}| = \mathbf{o}$. In contrast, Rule (Val-Rel-Int) relates integer pointers to arbitrary *invalid* handles.

The relation between source and target events $\alpha =_\delta \boldsymbol{\alpha}$, relates the same single events (Figure 10, bottom). When relating reads, writes, allocates, and frees, we insist that source pointers and target handles are related (according to the cross-language value relation) and the handles are valid (these rules have the validity bit set to **true**). Additionally, for reading and writing, they values being read or written must be of related types, i.e., $[\![\tau]\!] = \boldsymbol{\tau}$.

For memory-safe, well-typed source programs ($\vdash \mathsf{M} : \mathsf{wt}$), Theorem 2 states that the compiler produces equivalent memory-safe target programs; i.e., the compiled program emits a memory-safe trace that is related to the source trace.

Theorem 2 (Memory-Safety Preservation)**.**

$$\text{If } \vdash \mathsf{M} : \mathsf{wt} \text{ and } \mathsf{M} \twoheadrightarrow \alpha^* \text{ and } \mathtt{MS}(\alpha^*) \text{ then } \exists \delta, \boldsymbol{\alpha}^*.\ [\![\mathsf{M}]\!] \twoheadrightarrow \boldsymbol{\alpha}^* \text{ and } \alpha^* =_\delta \boldsymbol{\alpha}^* \text{ and } \mathtt{MS}(\boldsymbol{\alpha}^*)$$

In contrast, Theorem 3 states that memory-unsafe programs compiled to MSWasm abort at the first memory violation.

Theorem 3 (Memory Violations Trap)**.**

$$\text{If } \vdash \mathsf{M} : \mathsf{wt} \text{ and } \mathsf{M} \twoheadrightarrow \alpha^* +\!\!+ [\alpha] +\!\!+ \alpha'^* \text{ and } \mathtt{MS}(\alpha^*) \text{ and } \neg\mathtt{MS}(\alpha^* +\!\!+ [\alpha])$$
$$\text{then } \exists \delta, \boldsymbol{\alpha}^*.\ \alpha^* =_\delta \boldsymbol{\alpha}^* \text{ and } [\![\mathsf{M}]\!] \twoheadrightarrow \boldsymbol{\alpha}^* +\!\!+ [\mathbf{trap}]$$

Together these theorems characterize the scope of our compiler-based memory-safety *enforcement*:

Corollary 1 (Memory-Safety Enforcement)**.** If $[\![\mathsf{M}]\!] \twoheadrightarrow \boldsymbol{\alpha}^*$ then $\mathtt{MS}(\boldsymbol{\alpha}^*)$

Figure 11 shows the essence of the proof technique for Theorem 2, in the diagram, full arrows represent hypotheses and dashed arrows represent conclusions.

In the theorem statement, judgements of the form $M \twoheadrightarrow \alpha^*$ unfold to the reflexive-transitive closure of a single semantics step (i.e., the rules presented in Figure 4 for MSWasm and in Figure 8

$$\frac{\delta(\mathsf{b}, \mathsf{n_{id}}) = \mathbf{b}, \mathbf{n_{id}} \qquad \ell \times |\mathsf{w}| = \ell \qquad (\mathsf{a} - \mathsf{b}) \times |\mathsf{w}| = \mathbf{o}}{\mathsf{a}^{(\mathsf{b},\ell,\mathsf{w},\mathsf{n_{id}})} \sim_\delta \langle \mathbf{b}, \mathbf{o}, \ell, \mathbf{true}, \mathbf{n_{id}} \rangle} \text{ (Val-Rel-Ptr)} \qquad \frac{}{\mathsf{n} \sim_\delta \langle \mathbf{b}, \mathbf{o}, \ell, \mathbf{false}, \mathbf{n_{id}} \rangle} \text{ (Val-Rel-Int)}$$

$$\frac{[\![\tau]\!] = \tau \qquad \mathsf{a}^{(\mathsf{b},\ell,\mathsf{w},\mathsf{n_{id}})} \sim_\delta \langle \mathbf{b}, \mathbf{o}, \ell, \mathbf{true}, \mathbf{n_{id}} \rangle}{\mathsf{read}_\tau(\mathsf{a}^{(\mathsf{b},\ell,\mathsf{w},\mathsf{n_{id}})}) =_\delta \mathbf{read}_\tau(\langle \mathbf{b}, \mathbf{o}, \ell, \mathbf{true}, \mathbf{n_{id}} \rangle)} \text{ (Tr-Rel- Read-Ptr)}$$

$$\frac{[\![\tau]\!] = \tau \qquad \mathsf{a}^{(\mathsf{b},\ell,\mathsf{w},\mathsf{n_{id}})} \sim_\delta \langle \mathbf{b}, \mathbf{o}, \ell, \mathbf{true}, \mathbf{n_{id}} \rangle}{\mathsf{write}_\tau(\mathsf{a}^{(\mathsf{b},\ell,\mathsf{w},\mathsf{n_{id}})}) =_\delta \mathbf{write}_\tau(\langle \mathbf{b}, \mathbf{o}, \ell, \mathbf{true} \rangle)} \text{ (Tr-Rel-Write-Ptr)}$$

$$\frac{(\mathsf{a}^{(\mathsf{b},\ell,\mathsf{w},\mathsf{n_{id}})}) \sim_\delta \langle \mathbf{b}, \mathbf{o}, \ell, \mathbf{true}, \mathbf{n_{id}} \rangle}{\mathsf{salloc}(\mathsf{a}^{(\mathsf{b},\ell,\mathsf{w},\mathsf{n_{id}})}) =_\delta \mathbf{salloc}(\langle \mathbf{b}, \mathbf{o}, \ell, \mathbf{true}, \mathbf{n_{id}} \rangle)} \text{ (Tr-Rel-Allocate)}$$

$$\frac{(\mathsf{a}^{(\mathsf{b},\ell,\mathsf{w},\mathsf{n_{id}})}) \sim_\delta \langle \mathbf{b}, \mathbf{o}, \ell, \mathbf{true}, \mathbf{n_{id}} \rangle}{\mathsf{sfree}(\mathsf{a}^{(\mathsf{b},\ell,\mathsf{w},\mathsf{n_{id}})}) =_\delta \mathbf{free}(\langle \mathbf{b}, \mathbf{o}, \ell, \mathbf{true}, \mathbf{n_{id}} \rangle)} \text{ (Tr-Rel-Free)}$$

Fig. 10. Cross-language equivalence relation: values (top) and events (bottom).

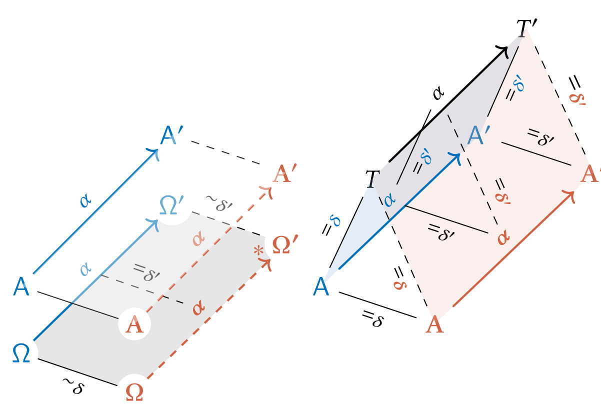

Fig. 11. Proof diagram for Theorem 2: functional correctness (left) and memory-safety preservation (right).

for C). The proof then proceeds unsurprisingly by induction over the reflexive-transitive reductions that generate the source trace, the figure shows the single-step case. We use metavariable Ω to indicate program states, which are the tuples presented in the semantics rules of each language.

We first describe the most interesting case of the functional correctness part of Theorem 2, i.e., the left of Figure 11. There, we need to show how one single source step ($\Omega \xrightarrow{\alpha}\!\!\!\!\twoheadrightarrow \Omega'$) that triggers a change in the source allocator ($\mathsf{A} \overset{\alpha}{\hookrightarrow} \mathsf{A}'$)[6], causes a series of 'related' target steps ($\mathbf{\Omega}' \xrightarrow{\alpha}\!\!\!\!\twoheadrightarrow^* \mathbf{\Omega}'$) that change the target allocator accordingly ($\mathbf{A} \overset{\alpha}{\hookrightarrow} \mathbf{A}'$). Essentially, target steps are related when they generate actions that are related (as per Figure 10), and they take related states ($\Omega \sim_\delta \mathbf{\Omega}$) into still-related states ($\Omega' \sim_\delta \mathbf{\Omega}'$). We do not present the formalisation of the state relation, intuitively it

[6]For ease of reading, we massage the allocator reduction judgement $\langle \mathsf{H}, \mathsf{A} \rangle \overset{\alpha}{\hookrightarrow} \langle \mathsf{H}', \mathsf{A}' \rangle$ to only contain the allocator.

just lifts the value relation of Figure 10 to all elements of a program state. Proving that the allocators step using related actions ensures that the source and target allocators are in related, consistent states. This is key to the memory safety preservation part of the theorem, i.e., the right of Figure 11.

To prove memory safety preservation for Theorem 2, we start from the functional correctness square between allocators (i.e., the base of the prism on the right in Figure 11). Then, we assume that the C step is memory-safe, this is represented by the blue side of the prism. Technically, this relies on another omitted piece of formalisation that relates source allocator states A and shadow memories T, a relation that holds when the addresses tracked in A and T are the same up to a bijection δ. The source memory safety assumption tells that the states of the initial C allocator and of the initial shadow memory are related (A $=_\delta T$), that they take a related step ($\alpha =_{\delta'} \alpha$), and that leads to related final states (A′ $=_{\delta'} T'$). The goal of the memory safety part of the proof is depicted as the corresponding red side of the prism: there is a relation between the states of the initial MSWasm allocator and of the initial shadow memory (**A** $=_\delta T$), the states take a related step ($\boldsymbol{\alpha} =_{\delta'} \alpha$) and that leads to related final states (**A′** $=_{\delta'} T'$). To construct the relations in the red square, we need to derive the dashed edges of the vertical triangles according to the correct relation with the correct bijection. This relation we obtain by combining the corresponding source-to-target relations (i.e., A $=_\delta$ **A**) and the source-to-monitor relation (i.e, A $=_\delta T$), and compose their bijections to relate abstract and target locations. That is, we obtain **A** $=_\delta T$, where $\boldsymbol{\delta}$ (relating MSWasm and abstract addresses) is the composition of δ (relating MSWasm and C addresses) with δ (relating C and abstract addresses). Importantly, the triangle of relations guarantees that the C notion of memory safety is preserved *exactly* in MSWasm. Since in C we instantiate our abstract notion of memory safety to account for intra-object safety, we get the same fine-grained memory safety notion preserved in MSWasm.

The proof of Theorem 3 is analogous. There, we use the same intuition presented above to simulate all actions of the memory-safe trace $\boldsymbol{\alpha}^*$ starting from their memory-safe counterparts in α^*. Then, at some point, starting from related states, C performs a memory-unsafe action α and MSWasm emits a **trap**. This proof is by case analysis over C memory safety violations, which we identify by the related abstract monitor getting stuck. In the proof, we relate these violations to a *failing* memory safety check in MSWasm, which causes the compiled program to **trap**, as expected.

6 IMPLEMENTING MSWASM

In this section we describe our prototype MSWasm compilation framework (Figure 12). We implement two compilers *of* MSWasm following the language semantics of Section 3. Our first compiler is an ahead-of-time (AOT) compiler from Wasm to executable machine code (Section 6.1); it demonstrates MSWasm's flexibility in employing different enforcement techniques, including both software-based enforcement and hardware-accelerated enforcement. Our second compiler is a compiler from Wasm to Java bytecode (Section 6.2); it demonstrates MSWasm's compatiblility with just-in-time (JIT) compilation. We also implement a compiler from C *to* MSWasm (Section 6.3), following the formal compiler model of Section 5. We describe these prototypes next.

Our prototype implementation of MSWasm extends the bytecode of Wasm with instructions to manipulate the segment memory as well as handles. In doing so, it takes a few shortcuts in the name of expediency—most notably, it replaces the existing Wasm opcodes for **τ.load** and **τ.store** with **τ.segload** and **τ.segstore**. A production MSWasm implementation would support both segment-based and linear-memory-based operations simultaneously, by using two-byte opcode sequences for **τ.segload** and **τ.segstore**.

6.1 Ahead of Time Compilation of MSWasm

To compile MSWasm bytecode to machine code, we build on the rWasm compiler [Bosamiya et al. 2022]. rWasm is a provably-safe sandboxing compiler from Wasm to Rust, and thus to high-performance machine code.[7] We extended rWasm to support MSWasm as follows. We modified rWasm's frontend to parse MSWasm instructions and propagate them through to later phases. We updated rWasm's stack analysis to account for MSWasm's new types and instructions (e.g., τ.segload and τ.segstore, which take a handle as argument). Finally, we updated rWasm's backend—the code generator, specifically—to implement MSWasm's instructions and segment memory.

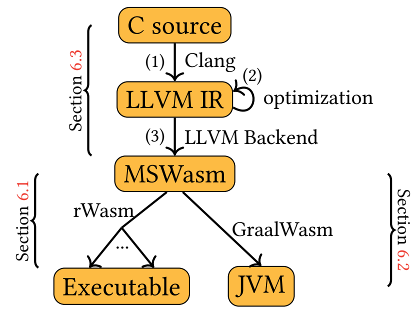

Fig. 12. End-to-end compilation pipeline. We first compile C to MSWasm (via LLVM), and then compile MSWasm to machine code using either our modified rWasm AOT compiler (which supports different notions of safety) or our modified GraalWasm JIT compiler.

One of the benefits of MSWasm is that it gives Wasm compilers and runtimes flexibility in how to best enforce memory safety. This is especially important today: memory-safety hardware support is only starting to see deployment and applications have different security-performance requirements—we cannot realistically expect everyone to pay the cost of software-based memory safety. When hardware becomes available, MSWasm programs can take advantage of hardware acceleration almost trivially: in our AOT compiler, for example, we only need to tweak the codegen stage. We demonstrate this flexibility by prototyping two different software techniques, and one hardware-accelerated technique that have different safety and performance characteristics.

Segments as Vectors. Our default technique for memory-safety enforcement closely matches Section 3.2.1, and enforces spatial safety, temporal safety, and handle integrity (rWasm$_{STH}$ in Section 7). We implement the segment memory as a vector (`Vec`) of segments. Each segment is a pair composed of a `Vec` of bytes (giving us spatial safety) and a `Vec` of tags, which is used to enforce handle integrity. Handles themselves are implemented using an `enum` (i.e., a tagged union). To enforce temporal safety we clear free segments from memory and use sentinel value to prevent the reuse of segment indexes. A slight variation of this technique (rWasm$_{ST}$ in Section 7) gives up on handle integrity (we remove the `Vec` of tags and related checks) for performance.

Segments with Baggy Bounds. Our second technique is inspired by Baggy Bounds checking [Akritidis et al. 2009], which is a technique that performs fast checks at each handle-modifying operation and elides checks at loads and stores, enabled by expanding buffers to the next power of two at the point of allocation. This technique gives up on handle integrity and temporal safety, since accesses are not checked, but is considerably faster (rWasm$_{S}$ in Section 7). To implement this technique, our compiler uses a single growable `Vec` of bytes, within which a binary buddy allocator allocates implicit segment boundaries. We implement the handles as 64-bit values storing an offset in memory and the log of the segment size (rounded up to nearest power of two at allocation). We emit bounds checks for each operation that might modify handles, ensuring that handles remain within the (baggy) bounds of their corresponding segment. Specifically, when handles stray a short

[7] In modifying rWasm, we were careful to ensure that we preserve its previously-established sandboxing/isolation guarantees. These guarantees, together with the internal memory-safety guarantees from MSWasm, increases the level of protection for native code generated by rWasm.

distance outside their segment, we mark them as such (and they can safely return back), but we trap when they (try to) stray too far.

Hardware acceleration using CHERI. Our third technique is to implement segment memory using CHERI capabilities [Watson et al. 2015]. Each handle is represented as a CHERI capability: a 128-bit register that consists of a capability base, offset, size, and permissions. Each capability also contains a tag bit that tracks the integrity of the capability—that the capability has not been expanded, has not been corrupted in memory, etc. By implementing handles using capabilities, we can enforce handle and spatial safety without software checks. This is because CHERI bounds checks memory accesses (spatial safety) and checks the validity of the tag bit (handle integrity) in hardware. Temporal safety for CHERI code is still a work in progress [Filardo et al. 2020; Xia et al. 2019], and consequently, our MSWasm-on-CHERI prototype does not yet support it.

To implement MSWasm using CHERI, we add a new rWasm backend that emits a subset of CHERI-compatible C code which implements handles as capabilities. Since CHERI capabilities and MSWasm handles are conceptually very similar, this compilation step is straightforward: `segalloc(size)` becomes `calloc(size, 1)`, `segfree(handle)` becomes `free(handle)`, `handle.add` becomes standard pointer addition, etc. We then compile this CHERI-C code to CHERI-Aarch64 using the already mature CHERI LLVM [CTSRD-CHERI 2022] compilation pipeline.

Implementation Effort. Our modifications to rWasm, for both software-only memory enforcement techniques, comprise roughly 1900 lines of additional code. The implementation of these two techniques comprise approximately 500 lines of code each in rWasm's codegen, and share the rest of rWasm's codebase. The hardware-accelerated technique also uses our modified rWasm frontend, but could not share much of rWasm's codegen with the other backends (which target Rust), instead requiring code to support a new target language—CHERI-C (even for the regular non-MSWasm-specific components of Wasm); our modifications for this backend comprise approximately 3000 lines of code. The relative ease of these modifications, both for software- and hardware-based techniques illustrates how MSWasm provides a fertile ground for experimenting with new techniques for providing performant memory safety.

6.2 Just in Time Compilation of MSWasm

Our second prototype is a just-in-time compiler of MSWasm built on top of GraalWasm [Prokopec 2019]. GraalWasm is a Wasm frontend for GraalVM [Oracle 2021a], a JVM-based JIT compiler capable of compiling a wide range of languages through the Truffle framework [Oracle 2021b]. We extend GraalWasm to support MSWasm. Our modifications mirror those we made to rWasm: We modified the GraalWasm frontend to parse MSWasm and the backend—the GraalWasm interpreter in this case—to support MSWasm's instructions and segment memory model. We were able to reuse the GraalVM JIT compiler unmodified, as it automatically optimizes the AST generated by Truffle from the interpreter.

Segments as Objects. Unlike our rWasm implementation, we only consider one enforcement technique. We pick a middle ground between safety and performance: We enforce spatial and temporal safety, but not handle integrity (Graal_{ST} in Section 7). Our implementation of memory segments in GraalWasm is similar to our first rWasm technique (but does not track handle-integrity tags). We implement the segment memory as a Java object, `SegmentMemory`, which tracks a list of segments. `SegmentMemory` is backed by Java's `Unsafe` memory manager, an internal framework that facilitates manual memory management. Unlike objects created on the Java heap, memory allocated through `Unsafe` is not garbage-collected and is accessed directly by pointer addresses. Using Java `Unsafe`, `SegmentMemory` manually allocates a new chunk of memory for each new segment, which lets us avoid the overhead of Java objects in exchange for explicitly tracking the allocated memory. A segment is represented by a `Segment` object, which contains an address within the `Unsafe` memory, the (inclusive) upper bound of the segment in memory, and a randomly generated key. To ensure temporal safety, free segments are removed from the list of segments in `SegmentMemory`, leaving no way to reference them.

Implementation Effort. We added roughly 1200 lines of code to GraalWasm. Our prototype is relatively simple and not yet tuned to take full advantage of GraalVM's optimizations. We leave this to future work.

6.3 Compiling C to MSWasm

MSWasm, like Wasm, is intended to be a compilation target from higher level languages. We implement a compiler from C to MSWasm by extending the CHERI fork of Clang and LLVM [CTSRD-CHERI 2022]. CHERI modified LLVM to support fat pointers, which share many characteristics with MSWasm handles, and is thus a good starting point for MSWasm.

CHERI represents fat pointers at the LLVM IR level as 64- to 512-bit pointers in a special, distinguished "address space"; pointers in this address space are lowered to CHERI capabilities in the appropriate LLVM backends. CHERI today only targets MIPS and RISC-V (with CHERI hardware extensions) backends; other backends, including the Wasm backend, are incompatible with CHERI's fat pointers. We modified the Wasm backend to emit MSWasm bytecode, lowering 64-bit fat-pointer abstractions to MSWasm abstractions. Since most of the implementation details follow from Section 5, we focus on details not captured by our formal model.

Global and Static Data. Our C-to-MSWasm compiler only emits **handle**-based load and store operations, resulting in MSWasm programs which do not use the linear memory at all. This provides additional safety guarantees (and implementation expediency) at the expense of some flexibility (e.g., we do not support integer-to-pointer casts, except for a few special cases like constant `0`). One consequence of this is that even global variables and static data need to be accessed via **handle**s, and thus placed in the segment memory.[8] Our compiler emits instructions to allocate a segment for each LLVM global variable and store the corresponding **handle** in a Wasm global variable. When the target program needs a pointer to the global array, it simply retrieves the **handle** from the appropriate Wasm global variable.

Some global variables in C are themselves pointers, initialized via initialization expressions, and need to be pointing to valid, initialized memory at the beginning of the program. Our compiler generates the necessary information in the output `.wasm` file to instruct MSWasm compilers and runtimes (e.g., rWasm and GraalWasm) to initialize certain segments at module initialization time.

C Stack. We compile part of the C stack to the segment memory. Specifically, stack variables whose address-of are taken and stack-allocated arrays cannot be placed on the (simple and safe) Wasm stack. Compilers from C to ordinary Wasm place these variables in the linear memory; our compiler places them in the segment memory.[9] We allocate a single large segment to represent stack memory for all of the variables which must be allocated in the segment memory; this means we have a single stack pointer, which we store in a dedicated Wasm global variable of type **handle**. Compared to using a separate segment for each stack allocation, our single-segment implementation

[8] More precisely, global variables which the program never takes the address of, do not need this treatment, as we can compile them into Wasm globals; but global variables which the program does take the address of, such as global arrays, are accessed via pointers and thus must be located in the segment memory.

[9] Stack variables which the program never takes the address of can be compiled to Wasm local variables, and data such as return addresses are never placed in the linear memory at all; Wasm implementations place them on a safe stack which is inaccessible to Wasm load and store instructions. The only stack variables which need to be placed in the linear memory, or for us the segment memory, are those we need pointers to.

is simpler (and faster) but trades-off some safety, e.g., we cannot prevent a stack buffer overflow from corrupting another stack-allocated buffer.

Standard library. Wasm programs which depend on `libc` need a Wasm-compatible implementation of `libc`. We modified WASI [WebAssembly [n.d.]] to be compatible with MSWasm to the extent necessary for our benchmarks. Most importantly, we fully recompiled the WASI `libc` using our MSWasm compiler, in order to generate `libc` bytecode compatible with MSWasm. In our MSWasm version of the WASI `libc`, the implementations of `malloc` and `free` are completely replaced by trivial implementations consisting of the segalloc and segfree MSWasm instructions.

Implementation Effort. Our CHERI LLVM additions (in particular to its Wasm backend) and the WASI `libc`, amounted to approximately 1600 lines of code. While our compiler can target any MSWasm backend, compiling general, real-world applications would likely require additional changes to WASI `libc`. We leave this to future work.

7 PERFORMANCE EVALUATION

In this section we describe our performance evaluation of the MSWasm compiler. We use the PolyBenchC benchmarking suite [Pouchet 2011] since PolyBenchC has become the de-facto suite used by almost all Wasm compilers (although limitations to PolyBenchC are noted by [Jangda et al. 2019]). We compare the performance of MSWasm to the performance of the same benchmarks compiled to normal Wasm, on each of our implementations.

Machine setup. We compile all benchmarks from C to Wasm using Clang, and from C to MSWasm using our modified CHERI Clang compiler; in both cases we set the optimization level to `-O3`. We run all our software-based enforcement benchmarks on a single core on a Linux-based system with an Intel Xeon 8160, and our hardware-accelerated enforcement benchmarks on the ARM Morello platform [ARM 2022].

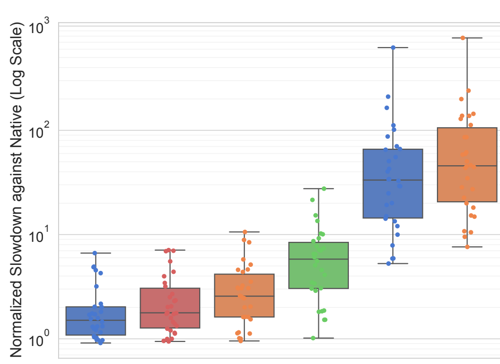

Fig. 13. Performance of our implementations of MSWasm compared to normal Wasm, normalized against native (non-Wasm) execution on benchmarks from PolyBenchC

Results. Figure 13 summarizes our measurements (see Appendix A for a detailed breakdown), normalized against the execution time of native (non-Wasm) execution. In this figure, $\text{rWasm}_{\text{Wasm}}$ and $\text{Graal}_{\text{Wasm}}$ refer to execution of normal Wasm. We distinguish the different MSWasm compilers according to their enforcement techniques: rWasm_{STH} enforces spatial safety, temporal safety, and handle integrity; rWasm_{ST} and Graal_{ST} only enforce spatial and temporal safety; and, rWasm_{S} only enforces spatial safety (in the style of baggy bounds).

As expected, and in line with prior work [Nagarakatte et al. 2009, 2010], each safety enforcement techniques comes with a performance cost—handle integrity being the most expensive. For the AOT compiler, we observe that enforcing spatial safety alone rWasm_S has a geomean overhead of 21.4% over $\text{rWasm}_{\text{Wasm}}$; additionally enforcing temporal safety (rWasm_{ST}) results in an overhead of 52.2% over $\text{rWasm}_{\text{Wasm}}$; and, finally, further enforcing handle integrity (rWasm_{STH}) increases the end-to-end overhead to 197.5%. For the JIT compiler, enforcing spatial and temporal safety results in an overhead comparable to that of the AOT compiler: Graal_{ST} imposes a 42.3% geomean overhead. The JIT approach is much slower than the AOT approach though—the overheads of $\text{rWasm}_{\text{Wasm}}$ and $\text{Graal}_{\text{Wasm}}$ over native (non-Wasm) execution are 71.8% and 3230.0% respectively. We also note that with increasing iterations of the GraalVM JIT, $\text{Graal}_{\text{Wasm}}$'s performance improves more rapidly than Graal_{ST}'s, which suggests that our implementation still has potential to make better use of GraalVM's optimizer.

Our hardware-accelerated approach, which enforces spatial safety and handle integrity (but not temporal safety), runs on an entirely distinct architecture and platform. Thus, a direct comparison against the same native code baseline used for the other techniques would not be particularly instructive. Instead, we evaluate our MSWasm-CHERI backend, against baseline native CHERI code (pure capability mode) on the Morello platform, and find an overhead of 51.7%.

Since normal Wasm and MSWasm have different bytecode formats, our evaluation of MSWasm performance necessarily includes slowdowns caused by inefficiencies in our compilation from C to MSWasm. But because MSWasm decouples memory safety enforcement from the generation of MSWasm bytecode, both parts of this pipeline (C-to-MSWasm compilation, and MSWasm to machine code) can be independently optimized, with MSWasm performance benefiting from improvements on both sides.

8 RELATED WORK

Comparison with [Disselkoen et al. 2019]. Their work introduces MSWasm and provides an informal design; however it only conjectures the memory-safety guarantees. In contrast, our current work specifies the design of MSWasm using formal semantics, which makes it possible to establish precise memory-safety guarantees, and to provide a specification for a variety of MSWasm implementations. Furthermore, we provide the first implementation and evaluation of MSWasm as well as a C-to-MSWasm compiler.

Memory safety for C-like languages. Despite a tremendous amount of work on memory-safety protection mechanisms [Szekeres et al. 2013], researchers still struggle to agree on a common definition for *memory safety* [Hicks 2014]. Azevedo de [Azevedo de Amorim et al. 2018] characterize memory safety as a 2-hypersafety property, similar to non-interference. Their definition belongs to a richer class of security properties, which are harder to enforce and to preserve robustly through compilation [Abate et al. 2019].

Many compiler-based instrumentations have been proposed to enforce memory safety in C programs via software-based checks attached to pointer and memory operations [Akritidis et al. 2009; Austin et al. 1994; Jim et al. 2002; Nagarakatte et al. 2009; Necula et al. 2005; Patil and Fischer 1997; Ruef et al. 2019; Xu et al. 2004]. Some of these solutions are also supported by formal memory-safety guarantees [Nagarakatte et al. 2009, 2010; Ruef et al. 2019]. These formal results however, are not *robust*, i.e., they do not guarantee memory safety when linking with arbitrary adversarial code. Moreover, these formalizations do not actually include the instrumentation pass of the compiler, but prove memory safety via *type safety* of an instrumented C-like language, where pointers are annotated with bounds metadata. Unlike MSWasm, these languages adopt a high-level memory model, which implicitly provides pointer integrity.

Our color-based memory-safety monitor and similarly our notion of authentic pointers and handles are inspired by previous work on *pointer provenance* in C [Memarian et al. 2019a]. Some of the C semantics proposed in that work track pointer provenance also through integer and pointer casts, which we do not consider in this work, also given that MSWasm has no native notion of casts. Our definition of memory-safety is also related to the micro-policies [d. Amorim et al. 2015]. The main difference is that they use (finite number of) color tags to enforce memory safety whereas we use (possibly infinite) colors to develop a general language-independent definition of memory safety.

Efficient memory-safety implementations. Unlike compiler-based instrumentations, compiling to MSWasm does not commit to a particular concrete strategy for enforcing memory safety: Different implementations of MSWasm can use different enforcement approaches. In particular, MSWasm enables backends compilers and runtimes to leverage efficient software- and hardware-based mechanisms, independently proposed to enforce pointer integrity [Liljestrand et al. 2019], spatial [Akritidis et al. 2009; Arm 2019; Kroes et al. 2018], and temporal [Lee et al. 2015; Parkinson et al. 2017] safety, to create new practical memory-safety enforcement schemes. Because MSWasm is platform-agnostic, we expect that implementations will be able to opportunistically take advantage of hardware memory protection mechanisms on individual platforms [Arm 2019; Devietti et al. 2008; Kwon et al. 2013; Oleksenko et al. 2018] (current and proposed) to efficiently implement handles.

Software isolation via Wasm. Wasm abstractions provide an efficient software-isolation mechanism, which has been applied in many different domains. For example, using Wasm, the RLBox framework [Narayan et al. 2020] retrofits isolation into the Firefox browser; Sledge [Gadepalli et al. 2020] enables lightweight serverless-first computing on the Edge; and eWASM [Peach et al. 2020] demonstrates practical software fault isolation for resource-constrained embedded platforms. These use cases already rely on both the performance and the sandboxing safety of Wasm, and stand to benefit from MSWasm's focus on memory safety.

[Bosamiya et al. 2022] use formal methods and non-traditional techniques respectively to provide provable isolation between the Wasm module, running as a native library, and the host process executing it. Their focus is on provable module–host isolation, and module-internal memory safety is explicitly left out of scope. As shown by [Lehmann et al. 2020], Wasm lacks many common defenses (e.g., stack canaries, guard pages, ASLR) against classic memory safety vulnerabilities, such as buffer overflows.

[Jangda et al. 2019] perform a large-scale performance evaluation of browser Wasm runtimes, comparing to native code. Our evaluation of MSWasm's performance (Section 7) shows that adding memory-safety protections does not fundamentally change Wasm's performance story. In particular, adding spatial and temporal safety imposes less overhead on Wasm than the overhead Wasm already incurs vs native code.

9 CONCLUSION

This paper realised the MSWasm proposal to extend Wasm with language-level memory-safety abstractions, giving it a formal semantics, proving that its programs are all memory safe and implementing the MSWasm language runtime. Like Wasm, MSWasm is intended to be used as a compilation target, so this paper formalised a C-to-MSWasm compiler, proved that it enforces memory safety, and implemented variations of said compiler with different tradeoffs between speed and security. Our PolyBenchC-based evaluation shows that MSWasm introduces an overhead ranging from 22% (enforcing spatial safety alone) to 198% (enforcing full memory safety). Our software-based implementations only serve to highlight that enforcing memory safety for Wasm is possible and, moreover, that MSWasm makes it easy to change the underlying enforcement mechanism without modifying application code. This means MSWasm engines will be able to take advantage of clever memory safety enforcement techniques today and hardware extensions in the near future, progressively (and transparently) improving the safety of the applications they run.

ACKNOWLEDGMENTS

This work was partially supported: by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762), by the Italian Ministry of Education through funding for the Rita Levi Montalcini grant (call of 2019) by a fellowship from the Alfred P. Sloan Foundation, and by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Gollamudi was supported in part through a generous gift to support research on applied cryptography and society in the Center for Research on Computation and Society (Harvard University) and Computing Innovating Fellowship (2021). We would also like to thank Reesha Rajen for their help on earlier stages of this work.

REFERENCES

Carmine Abate, Roberto Blanco, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *2019 IEEE 32th Computer Security Foundations Symposium (CSF 2019)*.

Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *USENIX Security Symposium*, Vol. 10.

Arm. 2019. Armv8.5-A Memory Tagging Extension. *White Paper* (2019).

ARM 2022. Arm Morello Program. https://www.arm.com/architecture/cpu/morello.

Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. 1994. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (Orlando, Florida, USA) *(PLDI '94)*. Association for Computing Machinery, New York, NY, USA, 290–301. https://doi.org/10.1145/178243.178446

Arthur Azevedo de Amorim, Cătălin Hriţcu, and Benjamin C. Pierce. 2018. The Meaning of Memory Safety. In *Principles of Security and Trust*, Lujo Bauer and Ralf Küsters (Eds.). Springer International Publishing, Cham, 79–105.

Jay Bosamiya, Wen Shih Lim, and Bryan Parno. 2022. Provably-Safe Multilingual Software Sandboxing using WebAssembly. In *USENIX Security Symposium*.

CTSRD-CHERI. 2022. The CHERI LLVM Compiler Infrastructure. https://github.com/CTSRD-CHERI/llvm-project.

A. A. d. Amorim, M. Dénès, N. Giannarakis, C. Hritcu, B. C. Pierce, A. Spector-Zabusky, and A. Tolmach. 2015. Micro-Policies: Formally Verified, Tag-Based Security Monitors. In *2015 IEEE Symposium on Security and Privacy*. 813–830. https://doi.org/10.1109/SP.2015.55

Thurston HY Dang, Petros Maniatis, and David Wagner. 2017. Oscar: A Practical page-permissions-based scheme for thwarting dangling pointers. In *USENIX Security*.

Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. 2008. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, New York, NY, USA.

Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. 2019. Position Paper: Progressive Memory Safety for WebAssembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM.

Nathaniel Wesley Filardo, Brett F Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, et al. 2020. Cornucopia: Temporal safety for CHERI heaps. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 608–625.

Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. 2020. Sledge: A Serverless-First, Light-Weight Wasm Runtime for the Edge. In *Proceedings of the 21st International Middleware Conference (Middleware '20)*. ACM.

Richard Grisenthwaite. 2019. Supporting the UK in becoming a leading global player in cybersecurity. https://community.arm.com/blog/company/b/blog/posts/supporting-the-uk-in-becoming-a-leading-global-player-in-cybersecurity.

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017a. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017b. Bringing the Web up to Speed with WebAssembly. *SIGPLAN Not.* 52, 6 (June 2017), 185–200. https://doi.org/10.1145/3140587.3062363

Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik Van Der Kouwe. 2016. TypeSan: Practical type confusion detection. In *Conference on Computer and Communications Security*. ACM.

Michael Hicks. 2014. What is memory safety? http://www.pl-enthusiast.net/2014/07/21/memory-safety/.

Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. ACM.

Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association.

Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *2002 USENIX Annual Technical Conference (USENIX ATC 02)*. USENIX Association.

Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2018. Delta Pointers: Buffer Overflow Checks without the Checks. In *Proceedings of the Thirteenth EuroSys Conference*. ACM.

Albert Kwon, Udit Dhawan, Jonathan M. Smith, Thomas F. Knight, and Andre DeHon. 2013. Low-Fat Pointers: Compact Encoding and Efficient Gate-Level Implementation of Fat Pointers for Spatial Safety and Capability-Based Security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification.. In *NDSS*. The Internet Society. http://dblp.uni-trier.de/db/conf/ndss/ndss2015.html#LeeSJWKLL15

Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association.

Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, et al. 2021. Cryptographic Capability Computing. In *IEEE/ACM International Symposium on Microarchitecture*. ACM.

Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. http://dx.doi.org/10.1007/s10817-009-9155-4

Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. 2019. PAC it up: Towards Pointer Integrity using ARM Pointer Authentication. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association.

Tyler McMullen. 2020. Lucet: A Compiler and Runtime for High-Concurrency Low-Latency Sandboxing. Principles of Secure Compilation (PriSC).

Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019a. Exploring C Semantics and Pointer Provenance. *Proc. ACM Program. Lang.* 3, POPL, Article 67 (Jan. 2019), 32 pages. https://doi.org/10.1145/3290380

Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. 2019b. Exploring C Semantics and Pointer Provenance. *Proc. ACM Program. Lang.* 3, POPL, Article 67 (jan 2019), 32 pages. https://doi.org/10.1145/3290380

Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the Depths of C: Elaborating the de Facto Standards. *SIGPLAN Not.* 51, 6 (June 2016), 1–15.

Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. *SIGPLAN Not.* 44, 6 (June 2009), 245–258. https://doi.org/10.1145/1543135.1542504

Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. *SIGPLAN Not.* 45, 8 (June 2010), 31–40. https://doi.org/10.1145/1837855.1806657

Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association.

George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe Retrofitting of Legacy Software. *ACM Trans. Program. Lang. Syst.* 27, 3 (May 2005), 477–526. https://doi.org/10.1145/1065887.1065892

Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX Explained: A Cross-Layer Analysis of the Intel MPX System Stack. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 2, Article 28 (jun 2018), 30 pages. https://doi.org/10.1145/3224423

Aleph One. 1996. Smashing the stack for fun and profit. *Phrack magazine* 7, 49 (1996), 14–16.

Oracle. 2021a. GraalVM. https://www.graalvm.org/.

Oracle. 2021b. Truffle Language Implementation Framework. https://www.graalvm.org/22.0/graalvm-as-a-platform/language-implementation-framework/.

Matthew Parkinson, Kapil Vaswani, Dimitrios Vytiniotis, Manuel Costa, Pantazis Deligiannis, Aaron Blankstein, Dylan McDermott, and Jonathan Balkind. 2017. *Project Snowflake: Non-blocking safe manual memory management in .NET*. Technical Report MSR-TR-2017-32. Microsoft. https://www.microsoft.com/en-us/research/publication/project-snowflake-non-blocking-safe-manual-memory-management-net/

Harish Patil and Charles Fischer. 1997. Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs. *Softw. Pract. Exper.* 27, 1 (jan 1997), 87–110.

Marco Patrignani. 2020. Why Should Anyone use Colours? or, Syntax Highlighting Beyond Code Snippets. *CoRR* abs/2001.11334 (2020). arXiv:2001.11334 https://arxiv.org/abs/2001.11334

Gregor Peach, Runyu Pan, Zhuoyi Wu, Gabriel Parmer, Christopher Haster, and Ludmila Cherkasova. 2020. eWASM: Practical Software Fault Isolation for Reliable Embedded Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3492–3505. https://doi.org/10.1109/TCAD.2020.3012647

Louis-Noel Pouchet. 2011. PolyBench-C: the Polyhedral Benchmark suite. https://web.cs.ucla.edu/~pouchet/software/polybench/. Accessed: March 2022.

Aleksandar Prokopec. 2019. Announcing GraalWasm – a WebAssembly engine in GraalVM. https://medium.com/graalvm/announcing-graalwasm-a-webassembly-engine-in-graalvm-25cd0400a7f2.

Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. 2019. Achieving Safety Incrementally with Checked C. In *Principles of Security and Trust*. Springer.

Fred B. Schneider. 2000. Enforceable Security Policies. *ACM Trans. Inf. Syst. Secur.* 3, 1 (feb 2000), 30–50. https://doi.org/10.1145/353323.353382

Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society.

Gang Tan. 2017. *Principles and Implementation Techniques of Software-Based Fault Isolation*. Vol. 1. Now Publishers Inc. 137–198 pages.

Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles (SOSP '93)*. ACM.

Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. 20–37. https://doi.org/10.1109/SP.2015.9

WebAssembly. [n.d.]. WebAssembly System Interface. https://github.com/WebAssembly/wasi.

Hongyan Xia, Jonathan Woodruff, Sam Ainsworth, Nathaniel W. Filardo, Michael Roe, Alexander Richardson, Peter Rugg, Peter G. Neumann, Simon W. Moore, Robert N. M. Watson, and Timothy M. Jones. 2019. CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 545–557. https://doi.org/10.1145/3352460.3358288

Wei Xu, Daniel C. DuVarney, and R. Sekar. 2004. An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*. ACM.

A DETAILED EVALUATION BREAKDOWN OF OUR IMPLEMENTATIONS OF MSWASM

Fig. 14. A detailed per-program breakdown of the performance of our implementations of MSWasm compared to normal Wasm, normalized against native (non-Wasm) execution on benchmarks from PolyBenchC. Rather than relying on less accurate external measurements using `time(1)`, we use PolyBenchC's own internal execution time reporting (i.e., `-DPOLYBENCH_TIME`).