Position Paper: Progressive Memory Safety for WebAssembly

Craig Disselkoen
UC San Diego
cdisselk@cs.ucsd.edu

John Renner
UC San Diego
jmrenner@eng.ucsd.edu

Conrad Watt
University of Cambridge
conrad.watt@cl.cam.ac.uk

Tal Garfinkel
Stanford University
talg@cs.stanford.edu

Amit Levy
Princeton University
aalevy@cs.princeton.edu

Deian Stefan
UC San Diego
deian@cs.ucsd.edu

ABSTRACT

WebAssembly (Wasm) is a low-level platform-independent bytecode language. Today, developers can compile C/C++ to Wasm and run it everywhere, at almost native speeds. Unfortunately, this compilation from C/C++ to Wasm also preserves classic memory safety vulnerabilities, such as buffer overflows and use-after-frees.

New processor features (e.g., tagged memory, pointer authentication, and fine grain capabilities) are making it increasingly possible to detect, mitigate, and prevent such vulnerabilities with low overhead. Unfortunately, Wasm JITs and compilers cannot exploit these features. Critical high-level information—e.g., the size of an array—is lost when lowering to Wasm.

We present MS-Wasm, an extension to Wasm that bridges this gap by allowing developers to capture low-level C/C++ memory semantics such as pointers and memory allocation in Wasm, at compile time. At deployment time, Wasm compilers and JITs can leverage these added semantics to enforce different models of memory safety depending on user preferences and what hardware is available on the target platform. This way, MS-Wasm offers a range of security-performance trade-offs, and enables users to move to progressively stronger models of memory safety as hardware evolves.

CCS CONCEPTS

• **Security and privacy → Web application security**; Hardware security implementation.

KEYWORDS

WebAssembly, Wasm, memory safety, tagged memory

1 INTRODUCTION

WebAssembly (Wasm) is a platform-independent bytecode designed to run C/C++ and similar languages at near native speed in the browser. Wasm's *linear memory model*—i.e., loads and stores to an untyped array of bytes, is the key feature that makes it possible for C/C++ compilers like Clang to easily and efficiently target Wasm. Unfortunately, this is also the reason memory safety vulnerabilities, like buffer overflows and use-after-frees (UAFs), remain a problem when C/C++ programs are compiled to Wasm [29, 32].

Wasm is designed to allow browsers to run code in a sandbox, isolating the impact of vulnerabilities in Wasm code from the rest of the browser.[1] But keeping the browser safe from Wasm code is not the same as keeping Wasm code safe from itself—isolation doesn't prevent attackers from exploiting memory-safety bugs to compromise the Wasm code and any data it handles.

This is worrisome. Wasm is supported by all major browser vendors, and already implemented in over 80% of all browsers on the web [8]. Wasm is also starting to find uses beyond the browser—from server-side runtimes [35, 44], to IoT platforms [17], and edge computing [18, 42]. As Wasm proliferates, we risk creating yet another ecosystem where memory-safety vulnerabilities are rampant.

Unfortunately, we can't simply modify Wasm to enforce strong memory safety by default, like past bytecodes for high-level languages (e.g., the Java bytecode or .NET common interface language). Requiring strong memory safety would be an anathema to the simplicity and performance that have fueled Wasm's broad adoption. Instead, we argue for a progressive approach to strong memory safety that neither mandates high performance overheads that could hinder widespread adoption, nor gives up on the goal of memory safety in the name of performance.

This progressive approach is both necessary and timely. As hardware acceleration makes memory safety increasingly cheap, the boundary between safety and high performance will narrow. For example, Sparc's application data integrity (ADI) [40] and ARM's upcoming memory tagging extension (MTE) [13] can probabilistically detect and prevent many buffer overflow and UAF bugs at near zero overhead—orders of magnitude faster than what's possible without dedicated hardware [40]. Similarly, ARM's recent pointer authentication feature can efficiently mitigate pointer corruption [26]. Looking further out, it seems likely that ARM will adopt some version of CHERI [15, 45] to efficiently enforce full *spatial safety* and eliminate

[1] For example, Wasm is type-safe, separates code and data, and enforces coarse-grained control flow integrity (CFI) [48]. All these design choices simplify isolation.

buffer overflow bugs altogether. Unfortunately, Wasm can't leverage these hardware features—too much high-level information is lost when compiling from C/C++ to Wasm's existing abstractions.

To bridge this gap, we propose Memory Safe WebAssembly (MS-Wasm), a backwards-compatible extension to Wasm that makes memory safety explicit at the language level. MS-Wasm extends Wasm's memory model with a new *segment memory* made up of *segments*—temporally safe extents of memory—and ensures that all accesses to the segment memory are via *handles*. Handles are strongly-typed first-class values that encapsulate bounds-checked, memory-safe pointers to the segment memory. With handles and segments, a C/C++ compiler can can encode all the semantics necessary to enforce *memory safety* [7]—in particular, *spatial safety*, *temporal safety*, and *pointer integrity*. By allowing handles to be *sliced*, MS-Wasm even captures fine-grained *intra-object* safety, e.g., to prevent a buffer in one field of a `struct` from overflowing into the next.

These richer semantics provide MS-Wasm backends—compilers and JITs—with everything they need to use different hardware and software approaches to ensure safety. It's then up to the backend to determine what policy to enforce based on the available hardware and needs of the user.

Some users might value detecting critical memory safety bugs in production but are unwilling to tolerate much overhead for enforcement—an MS-Wasm backend for ARM could use memory tagging (when available) to achieve this efficiently. On the other hand, a developer deploying a critical service, such as an authentication server, might value security more than performance, and thus request that the MS-Wasm backend enforce the full set of MS-Wasm safety properties, regardless of hardware support. A third user, e.g., a game developer unconcerned with security, might simply want to eschew any overheads and get performance equivalent to what normal Wasm would offer.

Our hope is that by ensuring memory safety overheads never exceed what is acceptable to the user, compiling with the semantics necessary for full memory safety will become the default. This can help incentivize the development of new hardware features: with MS-Wasm, if a vendor develops a new feature and changes a single JIT (e.g., V8 in Chrome) they can almost immediately expose its value billions of users. At the same time, as better hardware becomes widely available, MS-Wasm backends can seamlessly enable its use, providing a path to progressively better memory safety on the web, and other places where Wasm is used.

Organization. Next, we offer a brief overview of Wasm and its limited form of memory safety, then survey potential hardware features that could help (§ 2). We present MS-Wasm in § 4, and explore how it can improve memory safety for low-level languages like C/C++. In § 5 we discuss different hardware and software mechanisms MS-Wasm backends could leverage to enforce safety. Finally we discuss extensions to MS-Wasm and alternative design choices in § 6.

2 MOTIVATION

Our goal with MS-Wasm is to enhance Wasm for greater expressiveness, so that it can encode the semantics necessary to support different approaches to accelerating memory safety—more specifically, by adding a model of pointers and memory allocation so that this information isn't lost when lowering to Wasm.

To see why this is necessary, we will start by discussing the cause of memory safety vulnerabilities (§ 2.1); then sketch Wasm's basic structure, and why lowering to Wasm preserves these vulnerabilities (§ 2.2); and finally survey some of the current and future hardware support which MS-Wasm backends could use to prevent or mitigate memory-safety vulnerabilities (§ 2.3).

2.1 Memory Safety

In loose terms, memory-safety bugs in C/C++ result from how compilers interpret undefined behavior. For example, one valid interpretation of writing beyond the end of an array in the C standard is to crash the program—an easily understandable semantic. However, array bounds checking can induce unwanted performance overheads, so compilers adopt the more dangerous interpretation: write to whatever other object happens to be in that memory location, and continue running.

This interpretation violates programmers' assumptions about separation between different data objects [7]; and when this interpretation meets malicious inputs, they become memory-safety vulnerabilities, as the programmer has inadvertently given a potentially malicious input control over unintended parts of program data and control flow.

To prevent these attacks, the compiler could take a more conservative interpretation, and halt on undefined behaviors that violate separation—i.e., enforce *memory safety*. In practice, this amounts to ensuring three properties:

- *spatial safety*, which prevents out-of-bounds reads and writes;
- *temporal safety*, which prevents exploitation of use-after-free;
- *pointer integrity*, which prevents pointers from being manufactured from non-pointer values (e.g., casting an integer to a pointer), and also makes it impossible to corrupt a pointer in memory to create a different valid pointer.

Together, these three properties ensure that every pointer dereference in a program returns data from the corresponding, valid object.

Enforcing these properties efficiently requires some amount of dynamic checking—such as tracking if a pointer's referent has been de-allocated to prevent use-after-free bugs. Often the overhead of implementing these checks in pure software is prohibitive; even optimized JIT-based approaches can incur over 2× performance slowdowns for enforcing full memory safety [34].

Thus, the status quo for C/C++ has been to rely on system-level mitigations such as ASLR and W⊕X rather than enforce memory safety outright. Fortunately, hardware vendors are increasingly adding features to bring down the overhead of memory safety.

2.2 WebAssembly (Wasm)

Wasm is a portable bytecode language, designed to be an efficient target for low-level languages [16]. On the Web, developers use Wasm to embed existing C/C++ libraries such as the libsodium crypto library, video decoding libraries, and game engines into webpages. But Wasm's reach extends far beyond the browser. Server-side, Node.js application developers, for example, use Wasm to safely embed fast native code alongside JavaScript. Even serverless platforms (e.g., Fastly and Cloudflare) are making large bets on Wasm as the future of efficient, edge computing [18, 42].

Structurally, Wasm is a stack machine language that has well-typed stack (using simple primitive types: `i32`, `i64`, `f32`, and `f64`) and a linear model of memory, i.e., load and store to an "untyped array of bytes" [48] similar to native platforms. Consider, for example, a Wasm function that increments (by 3) a value in memory at a given address:

```
(func $add3 (param $addr i32)
  (i32.load (get_local $addr))
  (i32.add (i32.const 3))
  (i32.store (get_local $addr)))
```

This example shows how Wasm's values and stack operations are typed, but also how memory addresses simply have type `i32`; thus, loads and stores are free to arbitrarily read and write in Wasm's linear memory.

Because Wasm is designed to be an embedded in existing applications, Wasm code runs in an isolated sandbox. For example, even though Wasm code can access arbitrary indices in its own memory, there is no way for a Wasm instruction to access memory outside of its sandboxed area. The Wasm backend similarly protects return addresses with a separate stack and ensures that all indirect function calls go through well-typed entry points. Together, these protect Wasm code from stack-smashing attacks and make traditional return-oriented programming (ROP) attacks impractical.

Unfortunately, Wasm's safety is often misunderstood. For example, Wasm is sometimes called a "memory-safe language" [46]. This is not true: memory-safe languages provide *spatial safety*, *temporal safety*, and *pointer integrity* (§ 2.1); Wasm provides none of these.

Wasm, like native platforms, allows load and store instructions to operate on an untyped address space using arbitrary integer addresses. While attackers cannot carry out stack-smashing or ROP attacks, they can still exploit familiar memory-safety vulnerabilities in the Wasm linear memory to read and write data, just as they have in C/C++ applications on native platforms for decades. These attacks neither enable nor require escaping the Wasm sandbox. Indeed, compromising the Wasm application itself is often enough—plenty of sensitive data (e.g., cryptographic keys in the case of libsodium) is located within the sandbox.

2.3 Hardware Support for Memory Safety

Hardware tagged memory. Tagged memory associates additional metadata, a *tag*, with each region of memory. Research hardware-capability systems such as CHERI [45] and lowRISC [28] use tagged memory to ensure that capabilities cannot be forged or modified [21]. Other tagged memory systems such as Sparc ADI [40] associate, e.g., a 4-bit tag with each 64-byte aligned region of memory. In these systems, each pointer also contains a tag which is compared with the tag of the target memory on each load and store; if the tags don't match, the operation fails. The efficiency of this check makes memory tagging useful for enforcing a variety of protection policies [4]. For instance, memory tagging can be used for probabilistically detecting many spatial and temporal safety bugs.

ARM recently added a memory tagging extension (MTE) to the ARM 8.5-A ISA [13] which employ a 16-byte granule size but are otherwise nearly identical to ADI. This makes it likely that hardware tagged memory will be widely available in the near future.

Pointer authentication. ARM pointer authentication (PAC) is a feature which stores a cryptographic MAC in the unused upper bits of each pointer. PAC is supported by recent ARM processors, and already used in today's iPhones [38]. With PAC, memory operations can be made to fail if the pointer being dereferenced does not have a MAC from the appropriate private key (e.g., for kernel pointers, the kernel's private key) and context (an additional input to the MAC that can be used to provide compartmentalization). PAC instructions can be used to protect the integrity of data pointers, function pointers, and even stack pointers for CFI.

Bounds registers. Intel attempted to support memory safety with MPX *bounds registers* and *bounds tables*. With MPX, upper and lower bounds can be loaded into the bounds registers which are checked during loads and stores. Unfortunately, MPX failed to offer better performance than software-only solutions [36], leading to low adoption in practice, and even to GCC dropping support for MPX [12].

Hardware capabilities. The CHERI architecture [45] supports capabilities as an alternative to pointers. Capabilities are unforgeable references that contain both bounds data and access privileges, making them ideal for memory safety. While to date the CHERI architecture remains a research prototype, ARM recently announced plans to incorporate some of CHERI's ideas into future designs [15]. This represents a promising path forward for hardware memory safety.

3 DESIGN GOALS

We propose to extend Wasm to provide the capability to efficiently enforce full memory-safety guarantees, even inside the Wasm sandbox. The design of our extension, MS-Wasm, has four major goals:

Strong safety guarantees. MS-Wasm seeks to provide abstractions that can be used to enforce memory safety, i.e., *spatial* and *temporal safety*, and *pointer integrity*. At the same time, these abstractions should also be sufficient to support weaker piecemeal mitigation and detection mechanisms.

Backwards compatibility. MS-Wasm must be a minimally invasive extension to Wasm. This includes making MS-Wasm backwards compatible with existing Wasm toolchains, and making all of its features opt-in. Thus, existing Wasm binaries should remain valid, with the same semantics as before. Likewise, existing source-to-Wasm compilers should continue to be valid.

Leveraging hardware. MS-Wasm backends should be able to leverage whatever memory-safety hardware features are available on a given hardware platform. Thus, the design of MS-Wasm should be general enough to accommodate different detection and enforcement mechanisms, and not be specific to any particular hardware mechanism, e.g., memory tagging or MPX.

Progressive enforcement. Enforcing full memory safety is the ideal, but doing this without sufficient hardware support is prohibitive in many use cases—and requiring it would discourage users from building and deploying their applications with MS-Wasm.

Instead, MS-Wasm should accommodate different design points that trade off security and performance, and leave it to backends to choose the best combination of software and hardware mechanisms to implement the desired guarantees. For this reason, MS-Wasm separates the memory-safety *abstraction*—the Wasm-level semantics needed to allow C/C++ compilers to encode sufficient information

to efficiently enforce memory-safety guarantees—from the *enforcement policy*, i.e., how the backend actually implements whatever checks are necessary in order to meet the desired security and performance goals.

Different applications demand different points in the security-performance tradeoff space. For example, many applications would opt for mitigations over enforcement to stay within a reasonable performance budget (e.g., 5-10% CPU overhead). Security-critical applications, on the other hand, could demand full memory-safety guarantees, no matter the cost. Other applications might even request no enforcement at all—e.g., because performance is critical or, perhaps, because memory safety is enforced statically or dynamically, with inline checks. Such a policy could ideally be implemented with no overhead, equivalent to existing Wasm without any checks.

As hardware support improves and the cost of memory safety decreases, MS-Wasm backends will be able to provide progressively stronger guarantees at lower overheads, transparently increasing safety without violating users' performance requirements.

4 DESIGN

At the heart of MS-Wasm is a new *segment memory* that lives alongside the Wasm linear memory. Unlike the linear memory, the segment memory is well-structured; it consists of *segments*—linearly addressable, bounded regions of memory whose lifetimes are manually managed. Segments can only be accessed through *handles*, and not via Wasm's usual `load` and `store` instructions. This way, by placing certain restrictions on how handles are used, MS-Wasm can make strong guarantees about the memory safety of these segments.

In the rest of this section we describe MS-Wasm by showing how languages like C and C++ can be compiled to MS-Wasm, and how enforcing certain restrictions on MS-Wasm primitives can provide strong memory-safety guarantees.

4.1 Handles and Segments

Handles are used to model pointers—specifically, pointers bounded to particular live allocations of memory. Abstractly, handles are described by the 4-tuple (`base`, `offset`, `bound`, `isCorrupted`). The `base` of the handle represents the address of the start of the segment (in segment memory) being pointed to. The `offset` is the offset *within* the segment, i.e., within the bound, that the handle points to. If we think of the `handle` as a pointer, the location it points to in the segment memory is the `base+offset`.

As handles are used to model pointers, we introduce new Wasm instructions for pointer arithmetic, including addition, subtraction, and comparisons on handles; and we also define a `NULL` handle. For example, the `handle.add` and `handle.sub` instructions modify the handle offset without changing the base or bound. Pointer arithmetic can give rise to *out-of-bounds* handles (i.e., when the offset is negative or larger than the bound). We don't prevent code from creating such handles; instead, memory-safe MS-Wasm backends will trap when out-of-bounds handles are used, i.e., when the pointer is dereferenced. Delaying this check until dereference is important both for performance—it eliminates unnecessary checks during pointer arithmetic—and compatibility—as pointers that temporarily point out of bounds are common [11] and benign behavior in C programs [30, 31].

MS-Wasm treats handles as opaque values and does not specify a byte-level representation for them. This means that individual backends can represent them in a way most suitable to each platform, which may include storing some of this data separately and not as part of the handle itself. Moreover, as discussed below, backends which do not provide certain guarantees need not keep track of some of this data at all.

Segments are linearly addressable, fixed sized, extents of memory. (In § 4.3 we detail how segments are created and released.) Wasm code can load and store values to the segment memory via handles:[2]

```
i32.segment_load(src: handle) -> i32
i64.segment_load(src: handle) -> i64
i32.segment_store(dst: handle, val: i32)
i64.segment_store(dst: handle, val: i64)
```

To enforce spatial safety, an MS-Wasm backend must ensure the following property:[3]

> *Spatial safety for existing handles*: For each segment load and store, the handle being dereferenced is *in-bounds*, i.e., the handle is not the `NULL` handle, and its offset is nonnegative and less than its bound.

On the other hand, if these bounds checks are omitted by the backend, bounds information for handles need not even be tracked, and the performance of the segment load and store instructions should be similar to Wasm's existing load and store.

4.2 Slicing Handles

With the checks above, handles provide *inter-object* spatial safety, i.e., they restrict a pointer to accessing only the segment the handle points to. We also use handles to provide *intra-object* spatial safety through *slicing*. Wasm code can slice handles with:

```
segment_slice(parent: handle, base: i32,
              bound: i32) -> handle
```

This copies the parent handle and then grows the base, shrinks the bound, or both, to yield a smaller window into the segment. To illustrate this, consider the following code snippet.

```
struct A {char foo[4]; char bar[4];}
struct A * my_str = malloc(sizeof(struct A));
char * subfield = my_str->foo;
```

When we create a pointer to `foo` on the third line, the compiler can generate a new slice that includes only `foo`. Thus, if our code later tries to overflow `foo`, it will be contained to this slice by the spatial safety checks, and it will not be able to corrupt `bar`.

4.3 Segment Allocation and Deallocation

MS-Wasm code creates new segments and releases them with the new instructions:

[2] On notation: When practical, we adopt the Wasm convention of prefixing instruction names with their return type—for example, `i32.add`. When additional clarity is necessary, we adopt the notation `new_segment(size: i32) -> handle` which shows the arguments and return type explicitly.

[3] To fully prevent all out-of-bounds access to the segment memory, MS-Wasm backends must also provide the *handle integrity* property defined later, which prevents out-of-bounds handles from being forged by an attacker.

```
new_segment(size: i32) -> handle
free_segment(h: handle)
```

The `new_segment` instruction returns a handle to a newly-allocated segment. New segments are initialized to all zeroes, much how Wasm zero-initializes its linear memory [43]. These segments are guaranteed to be live until released with `free_segment`.

MS-Wasm backends enforcing memory safety should ensure *temporal safety*. We considered two different semantics for this. A simple approach would require memory-safe implementations to trap immediately when a segment is accessed after it has been freed. However, this is often inefficient to implement—it adds overhead to the critical path of `free_segment` and forces synchronization between the allocator and embedding application thread. Instead, we propose to adopt the relaxed model of Kedia et al. [22]:

> *(Relaxed) temporal safety*: The backend guarantees a trap on any access to a segment *after it has been deallocated*. That is, the segment may remain accessible (and completely valid) for an unspecified amount of time after `free_segment` has been called, until the allocator reclaims the memory.

This definition allows backends to defer deallocation until the last possible moment, while still preserving temporal safety. Moreover, it allows us to efficiently support several different safe manual memory management systems [5, 22, 27] as further discussed later (§ 5.3).

4.4 Handle Integrity

Since we model pointers with handles, code must be able to load and store handles from memory. Unlike C and C++, however, we do not provide a way to cast handles to integers (and back). This also means we cannot allow handles to be stored in the legacy Wasm linear memory (we simply do not provide instructions to do so). Instead, MS-Wasm provides instructions for explicitly loading and storing handles from segment memory:

```
handle.segment_load(src: handle) -> handle
handle.segment_store(dst: handle, val: handle)
```

To ensure that a Wasm program cannot forge pointers (e.g., with the `i64.segment_store` and `handle.segment_load` instructions), MS-Wasm backends should enforce handle integrity:

> *Handle integrity*: The backend conceptually associates either the type `data` or the type `handle` to each handle-aligned location in the segment memory. (New segments are initialized to be entirely type `data`.) On each `segment_load` and `segment_store` instruction, it then preserves these types:
>
> - Storing data (handle) to a particular location updates the containing segment element's type to `data` (`handle`).
> - Loading a handle from a location of type `data` produces a *corrupted handle*, i.e., a handle with `isCorrupted=True`. Like the `NULL` handle, corrupted handles are *invalid*—loads and stores on corrupted handles are disallowed; the backend should (conceptually) check the `isCorrupted` bit on every segment load and store. However, corrupted handles can themselves be written to segments with `segment_store`; when storing such a handle, the value of the loaded element is preserved, as is the type—`data`. This allows us to efficiently support `memcpy`-like operations (see § 6.1).
> - Loading data from a location of type `handle` is also allowed, but results in an implementation-defined data value. Importantly, the restrictions above ensure that such a data value can never be used as (or turned back into) a valid handle.

An MS-Wasm backend that does not enforce handle integrity need not keep track of `data`/`handle` types in the segment memory, and likewise need not distinguish corrupted handles from valid handles (need not keep track of `isCorrupted`); this should provide performance equivalent to existing Wasm. On the other hand, by enforcing handle integrity with the semantics described above, an MS-Wasm backend can ensure that *valid* handles can only be created during memory allocation or from existing valid handles with `segment_slice`, and furthermore that handles cannot be overwritten (even partially) in memory without becoming invalid.

5 IMPLEMENTATION STRATEGIES

MS-Wasm enables backend compilers and runtimes to enforce memory safety using a variety of hardware and software mechanisms. We review some of the promising current and future approaches.

5.1 Spatial Safety

To ensure memory safety, MS-Wasm backends must ensure that all dereferenced handles are in-bounds (§ 4.1).

In software. Enforcing full spatial safety in software often imposes relatively high overheads. For instance, Baggy Bounds [1] reported average runtime overheads around 60% (highly varying by workload) and memory overheads around 15%. ManagedC [14] reports full spatial safety with runtime overheads around 15% on average, but is highly workload dependent and relies on an optimized just-in-time compiler (JIT), resulting in additional memory overheads.

Lower overheads can be achieved by relaxing certain safety properties. Delta Pointers [23] achieves around 35% average runtime overhead and negligible memory overhead, partly by ignoring buffer underflow errors and only detecting buffer overflow. Going further, Duck et al. [10] report overheads similar to Baggy Bounds for full spatial safety, but by omitting certain checks (e.g., only checking writes), runtime overhead can be reduced below 10% [9]. However, even the fastest software schemes cannot match the efficiency possible with hardware support.

In hardware. The most promising and well-researched approach to strong spatial safety in hardware today is the Capability Enhanced RIsc (CHERI) system [45], which encodes spatial safety information in capability pointers—a fat pointer encoding that includes bounds information plus additional protection metadata. Current work suggests that overheads often in the low single digits are possible [6], and it seems likely that a production-quality processor could achieve even more impressive results. ARM recently announced plans to incorporate some of CHERI's ideas into future processors, and we feel optimistic about its prospects as the future of hardware-accelerated enforcement of full spatial safety.

Hardware tagged memory systems such as ARM MTE (§ 2.3) provide a weaker approach to spatial safety in the near term, but are very efficient. Using MTE, a MS-Wasm backend can ensure that the memory allocator assigns each allocation a different "color" (tag value) and ensure that adjacent allocations never share the same color. This is an incomplete solution, as many buffer overflows allow an attacker to address beyond adjacent objects. However, it does provide an efficient means of (probabilistically) detecting both spatial and temporal bugs. With 4-bit tags (as provided by ARM MTE), by randomly assigning a color to each allocation we can expect to detect both spatial and temporal bugs with a relatively high probability.

As MTE is only a mitigation, its security guarantees are not absolute, and it is unclear how much protection it provides against a determined attacker. For instance, an attacker may be able to brute-force the protection provided by randomly coloring allocations. In the end, the security benefits are highly dependent on details such as the particular type of vulnerability and how tags are assigned.

Finally, as discussed in § 2.3, Intel's MPX is already widely available, but is slower than comparable software solutions [36].

5.2 Handle Integrity

As specified in § 4.4, full memory safety requires MS-Wasm backends to track the type of memory in segments, either `handle` or `data`. This can also be done in either software or hardware.

In software. Efficiently implementing handle integrity in software is challenging: the overheads of software tagged memory systems such as ASan [39] suggest that both memory and CPU overheads can easily be prohibitive. With enough type information, it seems possible to do better—e.g., ManagedC [14] achieves surprisingly modest overheads for full memory safety with the help of full C type information. Extending MS-Wasm with additional semantics to support a scheme like this might be worth exploring in future work.

In hardware. Hardware tagged memory is the most direct and efficient way to support handle integrity. CHERI uses tags to distinguish between pointers and data, as does lowRISC [28].

Unfortunately, hardware tagged memory implementations such as Sparc's ADI and ARM's MTE cannot easily be used to provide handle integrity because they provide tags for 16-byte (MTE) or 64-byte (ADI) regions of memory only. This means that each region of this size must have a single tag at any given point in time. Even if it were practical to ensure that every 16-byte or 64-byte region of memory contained either only handles or only data at all times—which is far from clear—this would require coordination between Wasm compilers and backends to, e.g., provide proper padding for structs in C (as a Wasm backend could not easily redo this padding on its own). Moreover, MS-Wasm would have to choose a granularity for this padding independent of backend, which would either exclude certain platforms (if it were too small) or incur unnecessary space overheads (if it were too large). For tagged-memory systems to be useful for pointer integrity, they must provide tags (of at least one bit) at the granularity of pointer-size or smaller. At present, no commercial hardware tagged memory implementation does this.

ARM PAC (see § 2.3) seems like a natural fit for providing handle integrity. Unfortunately, it has some limitations. First, the overhead of using PAC to protect all pointers (as MS-Wasm proposes) is around 20% [26], much worse than what memory tagging can provide. Further, storing a MAC in the upper bits of each pointer makes these bits unavailable for use in the fat pointer encodings required for many spatial safety approaches (see § 5.1). Thus, although PAC may be a promising mitigation for protecting function or vtable pointers, it is not well-suited for providing handle integrity.

5.3 Temporal Safety

Temporal safety can be enforced in pure software, with the aid of virtual memory hardware, or using custom hardware designed for the purpose. MS-Wasm accommodates many recently proposed techniques by providing a separate segment memory and allocation interface (`new_segment` and `free_segment`) to support platform-specific allocators, and by slightly relaxing its temporal safety semantics (§ 4.3) to align with the guarantees these systems provide.

In pure software. Garbage collection is a traditional software-based solution for providing temporal safety. While an MS-Wasm implementation could employ garbage collection, recent systems provide substantially lower overheads while retaining manual memory management.

One approach, explored by DangSan [41] and other systems [25, 49], provides temporal safety by tracking all pointer aliases. When a pointer is freed, they rewrite its aliases to `NULL`. These systems still impose non-trivial overheads, e.g., DangSan incurs averages of 12%-41% runtime overhead, and 56%-140% memory overhead, on various single- and multi-threaded workloads (with results highly varying by workload, from 0% to over 700%). pSweeper [27] uses concurrent thread(s) to detect dangling pointers and avoids maintaining a precise points-to map; this results in lower runtime overheads than DangSan (12%-17% on average) with similar memory overheads. More efficient approaches are possible with help from the virtual memory system.

Using the virtual memory system. Both OSCAR [5] and Project Snowflake from MSR [22, 37] leverage the virtual memory system to efficiently provide temporally safe manual memory management. Project Snowflake tracks when most of the objects on a page have been freed, then unmaps the page (so that future dereferences of dangling pointers will trap) while copying the remaining live objects to a new page and lazily patching references to them. This is conceptually similar to a copying GC, but compared to GC, they reduced peak working set size by 3×, and runtime overhead by 2×. OSCAR maps a unique virtual page for each allocation and unmaps on each deallocation. So long as no virtual addresses are re-used, accesses to freed objects will hit an unmapped page. This provides strong temporal safety with very low runtime overhead. pSweeper, OSCAR, and Project Snowflake all leverage the delayed-free semantics of MS-Wasm.

In pure hardware. Watchdog [33] shows that dedicated hardware support can provide both temporal and spatial safety efficiently. It incurs a runtime overhead of 18%, and memory overheads averaging 32%-56% (again highly workload-variant), while providing full spatial and temporal safety.

ARM MTE style tagging can also be used to enforce temporal safety. Specifically, each memory allocation can be assigned a random tag when allocated, and re-tagged with a new random tag when freed. This has the potential to detect use-after-free bugs

efficiently enough to be used in production workloads. Unfortunately, this protection is probabilistic; an attacker could potentially easily brute-force this protection with, say, a 1/16 chance of success each time. Nonetheless, this presents an intriguing option in the security-performance tradeoff space which may be suitable for some applications.

6 DISCUSSION

In this section, we discuss some of the challenges with compiling MS-Wasm, how MS-Wasm can be used to further harden legacy code, alternative approaches to implementing memory safety for Wasm, and how future hardware mechanisms should shift to more efficiently support MS-Wasm and memory safety in general.

6.1 Compiling MS-Wasm

Handle sizing. We need to specify the size of handles at the Wasm level (independent of backends) so that compilers can target MS-Wasm. Handles are perhaps most naturally implemented as fat pointers, where bounds information is encoded directly in the pointer. We believe that 64-bit handles represent the optimal tradeoff: they are small enough to be efficient on modern architectures, while large enough to represent a handle (i.e., `base`, `bound`, `offset`) for Wasm's 32-bit address space when efficiently encoded [10, 23, 24]. Additionally, 64-bit handles are large enough to support most of the other schemes described above (for all safety properties).

Pointer semantics. The relationship between pointers and integer types in C is an important question for MS-Wasm which may impact compatibility with real-world C. The ISO C standard (§6.2.3.2) [19] is relatively strict as it defines casting integers to and from pointers as implementation-defined behavior or undefined behavior, with exception of `NULL` pointers (which MS-Wasm supports with the `NULL` handle). However, real-world C programs may rely on behavior beyond what is specified in the ISO standard.

With MS-Wasm the compiler can support some more liberal behaviors by modeling pointers with corrupted handles (see § 4.4) across casts in many situations, while MS-Wasm's strict rules on segment loads and stores ensure handle integrity is nevertheless always maintained. We believe our semantics maintains both broad compatibility with C programs and (for backends implementing the handle integrity checks) strong safety guarantees. An alternate semantics could allow more permissive casting between integers and handles while tracking provenance across integer types; however, this requires more invasive changes to Wasm's core semantics (e.g., to track `data`/`handle` types on the Wasm stack).

Another challenge is supporting functions like `memcpy()` and `memmove()` that can copy both data and pointers (e.g., by casting pointers to integers). But, since our `handle.segment_load` and `handle.segment_store` preserve the data loaded even when operating on a non-handle, they can be used to implement `memcpy()` and preserve all data and type information when copying segments.

6.2 Beyond Heap Memory Safety

Protecting the stack. MS-Wasm can be straightforwardly used to provide memory safety for heap allocations. However, the stack needs additional protection as well. C/C++-to-Wasm compilers like Emscripten [50, 51] currently *unsafely* store stack-allocated aggregate values like `struct` and `array` in linear memory, as Wasm stack and local variables can only hold scalars. To fully protect these values, we propose to store them in the segment memory. Much like Emscripten's current practice, the compiler could make a one-time large allocation for the entire stack. Then, when pointers to stack-allocated aggregate values are needed, the compiler can generate slices corresponding to those specific aggregates. These slices would not be temporally safe—they would remain valid after their target stack frame is popped—but would still be bounds checked and provide pointer integrity. Previous studies suggest that this tradeoff is reasonable, as they have failed to find exploitable use-after-free vulnerabilities on the stack [2, 25].

Protecting Wasm function pointers. The Wasm spec makes it clear that Wasm has no function pointers. Instead, functions are accessed through a table, ensuring that one can only branch to a function entry point. However, the indexes to this table still live in memory, and are thus vulnerable to control flow bending attacks [3].

Extending MS-Wasm to protect function pointers is a relatively simple change. We could add a `function_index` type that is in other ways like a normal `i32` function index, but leverages segments for integrity protection similar to handles. Like for handles, MS-Wasm backends could trap on any attempt to use a `function_index` that has been overwritten by a non-`function_index` value.

PAC is an ideal fit for this task, as prior work puts its overhead for protecting both code pointers and return addresses at (< 0.5%) on average [26].

6.3 Alternative Paths to Memory Safety

Memory-safe languages. One way to ensure Wasm programs are memory-safe would be to write them in a memory-safe language like Rust, or safe variants of C (e.g., [11, 20]). This, unfortunately, is not realistic—developers want to deploy legacy C/C++ code to Wasm—and even memory-safe languages can benefit from the hardware-accelerated memory safety provided by MS-Wasm.

Leaving enforcement to compilers. Another option is to put the onus for enforcing memory safety solely on compilers or language runtimes, and leave Wasm agnostic to these concerns. This again is unrealistic: many state-of-the-art techniques for efficient memory safety rely on architecture- or OS-specific features, from temporal safety techniques that leverage page level protections [5, 22], to hardware features like ARM PAC and MTE. For Wasm backends to efficiently leverage these features, we need abstractions to express memory-safety properties directly in Wasm.

Object-based memory models. One way to encode memory-safety properties in Wasm would be to extend Wasm with a strongly typed, object-based memory model that relies on garbage collection, like those in the JVM and CLR. These kinds of memory models offer both strong spatial and temporal safety (through garbage collection). Such a model has been proposed as an extension to Wasm to support higher level GC'd languages [47]. However, mapping C/C++ to this model seems to be fundamentally inefficient. Garbage collection brings additional space and compute overheads that are unnecessary in languages with manual memory management [37]. Moreover, the type systems of these languages are also too restrictive to model real C/C++ code [14].

6.4 Future Directions for Hardware

Web platforms have a long history of waiting until security problems are out of control before starting to address them. Memory safety vulnerabilities are the most common and dangerous vulnerabilities of our time. It is absurd to assume these issues will not impact Wasm. Now is the time to start addressing this challenge.

In the future, Wasm standards bodies could benefit from engaging with the architecture community on how to best surface future memory safety capabilities in Wasm, and ensure the Wasm road map takes these features into account. Conversely, future hardware designs could benefit from considering how to add value through surfacing memory features in the Wasm ecosystem. Providing a standard IR for memory safety, such as MS-Wasm, provides a target for hardware designers to work against and can play a critical role for enabling cooperation between these two communities.

ACKNOWLEDGMENTS

This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Watt was supported by the REMS EPSRC program grant (EP/K008528/1), and an EPSRC DTP award (EP/N509620/1). And the Gnome.

REFERENCES

[1] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. USENIX, 2009.
[2] J. Caballero, G. Grieco, M. Marron, and A. Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. ISSTA, 2012.
[3] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. USENIX, 2015.
[4] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA, 2007.
[5] T. H. Y. Dang, P. Maniatis, and D. A. Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. USENIX, 2017.
[6] B. Davis, R. N. M. Watson, A. Richardson, P. G. Neumann, S. W. Moore, J. Baldwin, D. Chisnall, J. Clarke, N. W. Filardo, K. Gudka, A. Joannou, B. Laurie, A. T. Markettos, J. E. Maste, A. Mazzinghi, E. T. Napierala, R. M. Norton, M. Roe, P. Sewell, S. Son, and J. Woodruff. CheriABI: Enforcing valid pointer provenance and minimizing pointer privilege in the POSIX C run-time environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, 2019.
[7] A. A. de Amorim, C. Hritcu, and B. C. Pierce. The meaning of memory safety. arXiv:1705.07354, 2017.
[8] A. Deveria. Can i use WebAssembly?, 2019. https://caniuse.com/#feat=wasm.
[9] G. J. Duck. LowFat. https://github.com/GJDuck/LowFat.
[10] G. J. Duck and R. H. C. Yap. Heap bounds protection with low fat pointers. CC, 2016.
[11] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi. Checked C: making C safe by extension. SecDev, 2018.
[12] GCC Wiki. Intel Memory Protection Extensions (Intel MPX) support in the GCC compiler. http://gcc.gnu.org/wiki/Intel%20MPX%20support%20in%20the%20GCC%20compiler, 2018.
[13] M. Gretton-Dann. Arm A-Profile architecture developments 2018: Armv8.5-A, Sep 2018. https://community.arm.com/processors/b/blog/posts/arm-a-profile-architecture-2018-developments-armv85a.
[14] M. Grimmer, R. Schatz, C. Seaton, T. Würthinger, and H. Mössenböck. Memory-safe execution of C on a Java VM. PLAS, 2015.
[15] R. Grisenthwaite. 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, 2019.
[16] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien. Bringing the Web up to speed with WebAssembly. PLDI, 2017.
[17] A. Hall and U. Ramachandran. An execution model for serverless functions at the edge. IoTDI, 2019.
[18] P. Hickey. Announcing Lucet: Fastly's native WebAssembly compiler and runtime, Mar 2019.
[19] Information technology – programming languages – C. Standard, International Organization for Standardization, June 2018.
[20] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. ATEC '02. USENIX Association, 2002.
[21] A. Joannou, J. Woodruff, R. Kovacsics, S. W. Moore, A. Bradbury, H. Xia, R. N. Watson, D. Chisnall, M. Roe, B. Davis, et al. Efficient tagged memory. In *2017 IEEE International Conference on Computer Design*, ICCD, 2017.
[22] P. Kedia, M. Costa, D. Vytiniotis, M. Parkinson, K. Vaswani, and A. Blankstein. Simple, fast and safe manual memory management. PLDI, June 2017.
[23] T. Kroes, K. Koning, E. van der Kouwe, H. Bos, and C. Giuffrida. Delta pointers: Buffer overflow checks without the checks. EuroSys, 2018.
[24] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. CCS, 2013.
[25] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee. Preventing use-after-free with dangling pointers nullification. NDSS, 2015.
[26] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J. Ekberg, and N. Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. *CoRR*, 2018.
[27] D. Liu, M. Zhang, and H. Wang. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping. CCS, 2018.
[28] lowRISC. lowRISC: A fully open-sourced, linux-capable, system-on-a-chip. https://www.lowrisc.org/.
[29] B. McFadden, T. Lukasiewicz, J. Dileo, and J. Engler. WebAssembly: A new world of native exploits on the browser. In *Blackhat briefings 2018*, 2018.
[30] K. Memarian, V. B. F. Gomes, B. Davis, S. Kell, A. Richardson, R. N. M. Watson, and P. Sewell. Exploring C semantics and pointer provenance. POPL, 2019.
[31] K. Memarian, J. Matthiesen, J. Lingard, K. Nienhuis, D. Chisnall, R. N. M. Watson, and P. Sewell. Into the depths of C: Elaborating the de facto standards. PLDI, 2016.
[32] M. Miller. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. https://www.youtube.com/watch?v=PjbGojjnBZQ, 2019. BlueHat.
[33] S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. ISCA, 2012.
[34] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. POPL '02. ACM, 2002.
[35] Node.js Foundation. Node.js. https://nodejs.org/en/, 2019.
[36] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer. Intel MPX explained: A cross-layer analysis of the Intel MPX system stack. *Proc. ACM Meas. Anal. Comput. Syst.*, 2(2), June 2018.
[37] M. Parkinson, K. Vaswani, M. Costa, P. Deligiannis, A. Blankstein, D. McDermott, J. Balkind, and D. Vytiniotis. Project Snowflake: Non-blocking safe manual memory management in .NET. Technical report, July 2017.
[38] Project Zero. Examining pointer authentication on the iPhone XS. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html, 2019.
[39] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. USENIX, 2012.
[40] K. Serebryany, E. Stepanov, A. Shlyapnikov, V. Tsyrklevich, and D. Vyukov. Memory tagging and how it improves C/C++ memory safety. *CoRR*, abs/1802.09517, 2018.
[41] E. van der Kouwe, V. Nigade, and C. Giuffrida. DangSan: Scalable use-after-free detection. EuroSys, 2017.
[42] K. Varda. WebAssembly on Cloudflare workers, Dec 2018.
[43] W3C. WebAssembly core specification, 2019. https://webassembly.github.io/spec/core/bikeshed/index.html#data-segments%E2%91%A0.
[44] Wasmer. Wasmer - universal WebAssembly runtime. https://wasmer.io/, 2019.
[45] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. SP, 2015.
[46] WebAssembly Community Group. WebAssembly, 2018. http://webassembly.org.
[47] WebAssembly Community Group. GC extension, 2019. https://github.com/WebAssembly/gc/blob/master/proposals/gc/Overview.md.
[48] WebAssembly Community Group. Semantics, 2019. https://github.com/WebAssembly/design/blob/master/Semantics.md.
[49] Y. Younan. FreeSentry: Protecting against use-after-free vulnerabilities due to dangling pointers. NDSS, 2015.
[50] A. Zakai. Emscripten: An LLVM-to-JavaScript compiler. OOPSLA, 2011.
[51] A. Zakai. Compiling to WebAssembly: It's Happening!, 2015. https://hacks.mozilla.org/2015/12/compiling-to-webassembly-its-happening/.