Iris-MSWasm: Elucidating and Mechanising the Security Invariants of Memory-Safe WebAssembly

MAXIME LEGOUPIL, Aarhus University, Denmark
JUNE ROUSSEAU, Aarhus University, Denmark
AÏNA LINN GEORGES, MPI-SWS, Germany
JEAN PICHON-PHARABOD, Aarhus University, Denmark
LARS BIRKEDAL, Aarhus University, Denmark

WebAssembly offers coarse-grained encapsulation guarantees via its module system, but does not support fine-grained sharing of its linear memory. MSWasm is a recent proposal which extends WebAssembly with fine-grained memory sharing via handles, a type of capability that guarantees spatial and temporal safety, and thus enables an expressive yet safe style of programming with flexible sharing. In this paper, we formally validate the pen-and-paper design of MSWasm. To do so, we first define MSWasmCert, a mechanisation of MSWasm that makes it a fully-defined, conservative extension of WebAssembly 1.0, including the module system. We then develop Iris-MSWasm, a foundational reasoning framework for MSWasm composed of a separation logic to reason about known code, and a logical relation to reason about unknown, potentially adversarial code. Iris-MSWasm thereby makes explicit a key aspect of the implicit universal contract of MSWasm: robust capability safety. We apply Iris-MSWasm to reason about key use cases of handles, in which the effect of calling an unknown function is bounded by robust capability safety. Iris-MSWasm thus works as a framework to prove complex security properties of MSWasm programs, and provides a foundation to evaluate the language-level guarantees of MSWasm.

CCS Concepts: • **Security and privacy → Logic and verification**; • **Theory of computation → Logic and verification**; *Higher order logic*; **Programming logic**; **Separation logic**; *Formalisms*.

Additional Key Words and Phrases: WebAssembly, Wasm, MSWasm, Capabilities, Memory Safety, Encapsulation, Logical Relation

ACM Reference Format:
Maxime Legoupil, June Rousseau, Aïna Linn Georges, Jean Pichon-Pharabod, and Lars Birkedal. 2024. Iris-MSWasm: Elucidating and Mechanising the Security Invariants of Memory-Safe WebAssembly. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 282 (October 2024), 29 pages. https://doi.org/10.1145/3689722

1 Introduction

WebAssembly (abbreviated Wasm) is the current industry standard to run applications efficiently in the browser [Haas et al. 2017], and is increasingly adopted in cloud computing (for example, Fastly's Compute@Edge [Fastly documentation 2022; Hickey 2020] and Fermyon's Spin [Butcher 2022]), in part thanks to its well-defined semantics and the high-performance implementations it enables. To rise up to the stringent security requirements of the web, Wasm promises not only sandboxing, but also several language-level security guarantees, including control flow integrity and coarse-grained

Authors' Contact Information: Maxime Legoupil, Aarhus University, Aarhus, Denmark, maxime@cs.au.dk; June Rousseau, Aarhus University, Aarhus, Denmark, june.rousseau@cs.au.dk; Aïna Linn Georges, MPI-SWS, Saarbrücken, Germany, algeorges@mpi-sws.org; Jean Pichon-Pharabod, Aarhus University, Aarhus, Denmark, jean.pichon@cs.au.dk; Lars Birkedal, Aarhus University, Aarhus, Denmark, birkedal@cs.au.dk.

ACM 2475-1421/2024/10-ART282
https://doi.org/10.1145/3689722

memory safety at the level of its units of code distribution, *modules*. Each module can define a *linear memory* (or several, in Wasm 2.0), which is private by default, but which the module can explicitly export. In that case, any other module can import it, and thereby access it unrestrictedly. This unusually strong encapsulation guarantee that a non-exported memory cannot be affected by other modules [Rao et al. 2023] makes edge computing practical and lightweight [Clark 2019]: one can safely compose a module with untrusted, potentially adversarial library modules to perform tasks (image compression, etc.) on separate memories. However, sharing is an all-or-nothing affair: a linear memory is either completely private, or all of it is shared with every module. As pointed out by Lehmann et al. [2020], this means that many of the classical attacks against memory unsafe languages, targeting a finer granularity, also work against programs that are not specifically written to take advantage of module isolation of WebAssembly.

Thus, to take advantage of the memory isolation guarantees of Wasm, programs require either invasive changes to fit WebAssembly's module system even though programs are typically not written directly in WebAssembly, or rely on extensive copying (which is the approach taken by the Component Model [The Bytecode Alliance 2023a,b]).

To address this lack of flexibility, Disselkoen et al. [2019] and Michael et al. [2023] propose Memory-Safe WebAssembly (abbreviated MSWasm), a conservative extension of WebAssembly with a mechanism for fine-grained memory sharing in the form of capabilities [Dennis and Van Horn 1966; Wilkes and Needham 1979], which it calls *handles*, and which embody authority over ranges of a new kind of memory: *segment memory*. This design is inspired by the capability-enhanced CHERI hardware architecture [Woodruff et al. 2014], which has been shown to be targetable from C with lightweight code changes by relying on reasonable patches to production compilers [Memarian et al. 2016; Zaliva et al. 2024]. The expectation is that MSWasm programs respect much finer memory safety invariants than plain Wasm. However, as illustrated during the development of the CHERI capability hardware architecture, these security invariants are very brittle: a mistake in a single detail can invalidate all encapsulation guarantees [Bauereiss et al. 2022; Nienhuis et al. 2020], and prose specifications backed by mere testing do not provide the required level of assurance.

Contributions. In this paper, we complete the pen-and-paper definition of MSWasm to be a conservative extension of WebAssembly 1.0, and mechanise it in the Coq proof assistant as MSWasmCert, building on WasmCert [Watt et al. 2021]. On top of this precise language definition, we develop Iris-MSWasm, a program logic that extends Iris-Wasm [Rao et al. 2023] with capability reasoning. Using the assertion language of Iris-MSWasm, we formulate an unstated yet key part of the *universal contract* [Van Strydonck et al. 2019] of MSWasm: that all instructions respect *robust capability safety*. Robust capability safety, as demonstrated for object capabilities [Devriese et al. 2016; Swasey et al. 2017] and capability hardware architectures [Georges 2023; Georges et al. 2021a, 2022a, 2021b, 2022b; Skorstengaard 2019; Skorstengaard et al. 2018, 2019a,b], makes it tractable to reason about the combination of known code with unknown, potentially adversarial code. As such, it refines the original *memory safety* guarantee of MSWasm, which does not directly lend itself to prove integrity properties of local state.

With our definition in hand, we identify cases where the original prose description is imprecise, as well as a handful of minor typos. We then show that MSWasm satisfies robust capability safety, and illustrate it on key representative examples capturing fine-grained memory invariants, thereby validating the design of MSWasm to the level of rigour that it deserves. To our knowledge, this is the first proof of robust capability safety for an industrial language, and for a language of this size. Moreover, because our formulation of robust capability safety captures the behaviour of an arbitrary MSWasm module given the exports that the module has access to, we expect that it can be

used to reason about the combination of WebAssembly code compiled from a higher-level language with unknown code compiled to MSWasm.

In showing robust capability safety for a complete definition of MSWasm, we make the case that, in addition to the extensional behaviour of a formally defined operational semantics, *industrial-scale language definitions can and therefore should come with a formally stated universal contract backed by a machine-checked proof.*

In summary, our contributions are:

- MSWasmCert, a mechanised language definition of MSWasm.
- Iris-MSWasm, a mechanised program logic covering all the language constructs of MSWasm, and with a complete proof of soundness.
- A mechanised statement and proof of robust capability safety using a logical relation.

All the technical results have been proved in the Coq proof assistant, and our Coq development is available online (see Data Availability Statement).

Outline. In the rest of this section, we present capabilities/handles, illustrate their use on a running example (§1.1), and describe the attacker model that we consider (§1.2). We then present our precise semantics of MSWasm (§2), focusing on the differences to plain WebAssembly, and highlight how we complete the original prose semantics. We then describe our program logic and its assertion language (§3), which we then use for the main contribution of this paper: the definition and proof of robust capability safety of MSWasm (§4). We illustrate this property on a larger example (§5), and we finish with a discussion (§6).

1.1 Introduction to MSWasm via a Running Example

We illustrate MSWasm on the classic capability 'buffer' example [Woodruff et al. 2023], adapted to our setting. We give the code (using the formal syntax we present later in Figure 2) and depict it visually in Figure 1, and describe it informally below.

```
0 [ i32.const   8;          } Allocate a handle
1   segalloc;               } with room for two i32s
2   local.set   $h;           Call it $h
3   local.get   $h;         }
4   i32.const   42;         } Store private value 42
5   i32.segstore;           } at h[0..3]
6   local.get   $h;         }
7   i32.const   4;          } Create a sub-handle for
8   i32.const   4;          } the rest of the buffer, h[4..]
9   slice;                  }
10  local.set $hpub;          Call it $hpub
11  local.get $hpub;        } Call adversary function $adv
12  call      $adv;         } with argument $hpub
13  local.get   $h;         }
14  i32.segload   ]         } Read from $h
```

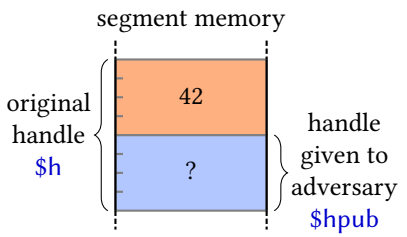

Fig. 1. The buffer example

The known code starts (lines 0–2) by allocating a handle that has authority over a 'buffer': a part of segment memory. It stores (3–5) a private value, 42, in the first four bytes. The intent is to call an untrusted function, $adv, with access to the rest of the buffer, but not to the private value. To do

so safely, the known code *slices* (6–9) the handle to get a sub-handle that has authority only over the rest of the buffer. The known code then calls $adv, sharing only the sub-handle by passing it as an argument (11–12), and finally reads back the private value (13–14).

MSWasm guarantees that the handle $h has not been freed and the private value is unchanged after the call to $adv returns. In general, MSWasm guarantees fine-grained memory safety: unless explicitly given access to a handle with authority over a part of segment memory, a module cannot read or write to that part of segment memory.

In the rest of the paper, we show how to prove that this program's return value is either the **trap** failure value (in case the allocation or adversary call traps), or 42. We use a program logic to reason about the known code, and a logical relation to reason about the unknown code.

In §5, we also illustrate this approach on a stack module that showcases MSWasm and demonstrates that our approach scales to complex invariants about practical data structures.

1.2 Attacker Model and TCB

Wasm modules are linked together via *instantiation*. Instantiation does not take place within a Wasm program, but in a *host* — in the browser, this is typically JavaScript code. Instantiation enforces that all the modules are well-typed and have consistent exports and imports. The attacker model that we consider is one where one or more 'friendly' modules with known code are instantiated with one or more unknown, potentially adversarial Wasm modules. We assume that the host does not affect memories, locals, control flow, etc.; in our formalisation, we do this by restricting the host language. This attacker model fits the context of cloud computing (microservices, edge computing, etc.), where one client's module should be isolated from the third-party libraries it imports.

Our results concern the language specification, not a particular implementation in term of a Wasm runtime, which we still have to trust. We prove integrity, but not confidentiality — this could be tackled using a binary logical relation expanding our unary logical relation [Georges 2023, §4.5], but it is outside of our scope to define an operational semantics that faithfully captures confidentiality in the setting of WebAssembly. We also have to trust the host language to match the assumptions stated above. On the mechanisation side, we have to trust the soundness of the 'kernel' proof checker of the Coq proof assistant. Crucially, we do not need to trust the Iris separation logic framework, nor the separation logic rules we define, as they are linked to the operational semantics of MSWasm by our adequacy theorem (§3.4).

2 The MSWasmCert Semantics

Michael et al. [2023] present MSWasm as an extension of WebAssembly. While their pen-and-paper specification of MSWasm builds on a mostly faithful representation of WebAssembly, it remains an idealised version of the language. This results in a language specification that does not exactly line up with the official language specification of WebAssembly. Meanwhile, unlike most other industrial languages, one of the advantages of WebAssembly is that it has a detailed and comprehensive semantics [Haas et al. 2017], with a well-defined standard [Rossberg 2019]. One of our goals is thus to formalise the MSWasm proposal as an extension of the *official* and *complete* WebAssembly semantics. This is achieved by building our formalisation on top of the WasmCert mechanisation, which covers the full language as per the 1.0 specification.

2.1 Plain Wasm Semantics

In this section, we briefly recall WebAssembly, highlighting the features omitted by Michael et al. [2023]; a reader familiar with the language can safely skip to §2.2. Figure 2 shows the syntax of WebAssembly, with the additions brought by MSWasm highlighted in magenta.

$$
\begin{array}{rl}
\text{(numeric type) } nt & ::= \mathbf{i32} \mid \mathbf{i64} \mid \mathbf{f32} \mid \mathbf{f64} \\
\text{(value type) } t & ::= nt \mid \mathbf{handle} \\
\text{(value) } v & ::= nt.\mathbf{const}\ c \mid \mathbf{handle.const}\ h \\
\text{(byte tag) } tag & ::= \mathbf{Numeric} \mid \mathbf{Handle} \\
\multicolumn{2}{c}{\text{(in the original presentation, these are called } \bigcirc \text{ and } \square\text{)}}
\end{array}
\qquad
\begin{array}{rl}
\text{(function type) } ft & ::= ts \to ts \\
\text{(immediate)} & \\
i, min, max, addr, off, id & ::= \mathbf{N} \\
\text{(tagged byte) } tbyte & ::= byte \times tag
\end{array}
$$

$$\text{(handles) } h ::= \{\mathbf{base} : addr,\ \mathbf{offset} : off,\ \mathbf{bound} : off,\ \mathbf{valid} : \mathbf{bool},\ \mathbf{id} : id\}$$

$$
\begin{array}{rl}
\text{(basic instructions) } b ::= & nt.\mathbf{const}\ c \mid t.\mathbf{add} \mid other\ stackops \mid \mathbf{local}.\{\mathbf{get/set}\}\ i \mid \mathbf{global}.\{\mathbf{get/set}\}\ i \mid \\
& t.\mathbf{load}\ (tp\_sx)^?\ a\ o \mid t.\mathbf{store}\ tp^?\ a\ o \mid \mathbf{memory.size} \mid \mathbf{memory.grow} \mid \\
& \mathbf{block}\ ft\ bs \mid \mathbf{loop}\ ft\ bs \mid \mathbf{if}\ ft\ bs\ bs \mid \mathbf{br}\ i \mid \mathbf{br\_if}\ i \mid \mathbf{br\_table}\ is \mid \mathbf{call}\ i \mid \\
& \mathbf{call\_indirect}\ i \mid \mathbf{return} \mid t.\mathbf{segload} \mid t.\mathbf{segstore} \mid \mathbf{segalloc} \mid \mathbf{segfree} \mid \\
& \mathbf{handle.add} \mid \mathbf{slice}
\end{array}
$$

(the flags of the **load** and **store** instructions represent a packed type, an alignment value and an offset.
The new **segload** and **segstore** instructions do not have similar flags)

$$\text{(administrative instructions) } e ::= b \mid \mathbf{handle.const}\ h \mid \mathbf{trap} \mid \mathbf{invoke}\ i \mid \mathbf{label}_i\{es\}\ es\ \mathbf{end} \mid \mathbf{local}_i\{F\}\ es\ \mathbf{end} \mid \mathbf{call\_host}\ tf\ hidx\ vs$$

$$
\begin{array}{rlrl}
\text{(functions) } func & ::= \mathbf{func}\ i\ ts\ bs & \text{(tables) } tab & ::= \mathbf{tab}\ min\ max \\
\text{(memories) } mem & ::= \mathbf{mem}\ min\ max & \text{(globals) } glob & ::= \mathbf{glob}\ \mathrm{mutable}\ t\ b_{\mathsf{init}} \\
\text{(elem segments) } elem & ::= \mathbf{elem}\ i\ bs_{\mathsf{off}}\ is & \text{(data segments) } data & ::= \mathbf{data}\ i\ bs_{\mathsf{off}}\ bytes
\end{array}
$$

$$
\begin{array}{rl}
\text{(import descriptions) } importdesc & ::= \mathbf{func_i}\ i \mid \mathbf{tab_i}\ min\ max \mid \mathbf{mem_i}\ min\ max \mid \mathbf{glob_i}\ \mathrm{mutable}^?\ t \\
\text{(imports) } import & ::= \mathbf{import}\ string\ string\ importdesc \\
\text{(export descriptions) } exportdesc & ::= \mathbf{func_e}\ i \mid \mathbf{tab_e}\ i \mid \mathbf{mem_e}\ i \mid \mathbf{glob_e}\ i \\
\text{(exports) } export & ::= \mathbf{export}\ string\ exportdesc \\
\text{(start) } start & ::= \mathbf{Some}\ i \mid \mathbf{None}
\end{array}
$$

$$
\begin{array}{rl}
\text{(function instances) } finst & ::= \{(inst; ts); es\}_{tf}^{\mathsf{NativeCl}} \mid \{hidx\}_{tf}^{\mathsf{HostCl}} \\
\text{(table instances) } tinst & ::= \{\mathsf{elem} : is,\ \ \mathsf{max} : max^?\} \\
\text{(memory instance) } minst & ::= \{\mathsf{data} : bytes,\ \ \mathsf{max} : max^?\} \\
\text{(global instance) } ginst & ::= \{\mathsf{mut} : mutable^?,\ \ \mathsf{value} : v\} \\
\text{(segment instance) } sinst & ::= \{\mathsf{segdata} : tbytes,\ \ \mathsf{max} : max^?\} \\
\text{(allocator instance) } ainst & ::= id \to (addr \times off)^? \\
\text{(store) } S & ::= \left\{\begin{array}{l}\mathsf{funcs} : finsts,\ \ \mathsf{globs} : ginsts,\ \ \mathsf{mems} : minsts,\ \ \mathsf{tabs} : tinsts, \\ \mathsf{seg} : sinst,\ \ \mathsf{allocator} : ainst\end{array}\right\} \\
\text{(frame) } F & ::= \{\mathsf{locs} : vs,\ \ \mathsf{inst} : inst\} \\
\text{(module instance) } inst & ::= \{\mathsf{types} : fts,\ \ \mathsf{funcs} : is,\ \ \mathsf{globs} : is,\ \ \mathsf{mems} : is,\ \ \mathsf{tabs} : is\} \\
\text{(modules) } m & ::= \left\{\begin{array}{l}\mathsf{types} : fts,\ \mathsf{funcs} : funcs,\ \mathsf{globs} : globs,\ \mathsf{mems} : mems,\ \mathsf{tabs} : tabs, \\ \mathsf{data} : datas,\ \mathsf{elem} : elems,\ \mathsf{imports} : imports,\ \mathsf{exports} : exports, \\ \mathsf{start} : start\end{array}\right\}
\end{array}
$$

Fig. 2. WebAssembly Abstract Syntax in black, with the MSWasm additions in magenta

A Stack Language. WebAssembly code is given as a list of instructions, and its operational semantics works as a stack machine that reduces the head instruction. For example, the operational semantics rule for addition is defined as

$$(S, F, [\mathbf{i32.const}\ c_1;\ \mathbf{i32.const}\ c_2;\ \mathbf{i32.add}]) \hookrightarrow (S, F, [\mathbf{i32.const}\ (c_1 + c_2)])$$

(we explain S and F below). In order to apply this rule in the context of a larger program, WebAssembly provides structural rules that allow to reduce under a context. For example, if $(S, F, es) \hookrightarrow (S', F', es')$, then $(S, F, vs +\!\!+ es +\!\!+ es_2) \hookrightarrow (S', F', vs +\!\!+ es' +\!\!+ es_2)$, where we write vs for a list of *values*, and es for a list of *expressions*.

The Store and the Frame. WebAssembly operates over a *store* and a *frame*. The *store* S is a record that bookkeeps all globally available functions, memories, global variables, etc. The *frame* F contains the current function's local variables, as well as its *instance*. The instance symbolises the function's environment, describing which parts of the global store the function has access to. It is defined as a record which contains indices that refer to objects in the store. This means that functions access the store via a level of indirection through the frame.

The key role of the instance in the frame is visible for example in the **global.get** instruction:

$$\frac{F.\mathsf{inst.globs}[i] = k \qquad S.\mathsf{globs}[k].\mathsf{value} = v}{(S, F, [\textbf{global.get}\ i]) \hookrightarrow (S, F, [v])}$$

All WebAssembly variables, as well as functions and all other objects are referred to with indices instead of names. For local variables, this index refers to the place in the list of local variables present in F.locs. However, for all other objects, because they may outlive the current function (and even the current module if they are exported), the value is kept in the store together with that of objects defined in other modules. The index into the store has to be looked up in the instance, F.inst. In the case of a global variable shown above, the instance's globs field is a list of indices into the store, the $i-$th of which corresponds to the location in the store of this module's $i-$th global variable. It is then from that location that we fetch the value of the variable from the store.

This indirection into the store via the frame is the crux of the coarse-grained encapsulation guarantees of WebAssembly. As we discuss in §2.2, handles achieve encapsulation very differently: they access the store directly, but are guarded by dynamic checks. The original presentation of MSWasm omits the instance from their description of the frame, and thus only accounts for handles. Meanwhile, our mechanisation captures both the coarse-grained encapsulation guarantees of WebAssembly, and the new fine-grained dynamic guarantees of handles.

Modules and Host Language. Frames and instances are constructed at runtime. Statically, WebAssembly code is shipped in *modules*, each module defining functions, a linear memory, global variables, etc. A module can *import* any of those objects, either from another WebAssembly module that explicitly *export*ed it, or from the *host language* that runs the WebAssembly modules.

Static modules are turned into dynamic module instances via *instantiation*, in which the module's code is typechecked, its imports are satisfied, and its exports are prepared for subsequent imports. This process is not part of WebAssembly itself, and hence WebAssembly code always runs embedded in a *host language*, typically Javascript, that performs module instantiation and can also perform an array of other interactions with WebAssembly code, such as calling WebAssembly functions, accessing or modifying WebAssembly state, etc. The host language can also provide functions or other objects that WebAssembly modules can import. As with frame instances, the original MSWasm presentation omits any description of modules and module instantiation.

Linear Memory. One of the objects that a module can encapsulate is the *linear memory*. In WebAssembly, the *linear memory* of a module (which Michael et al. [2023] call *heap memory*) is a growable array of bytes. Linear memory is accessed via **load** and **store** instructions, which take an **i32** argument from the stack and treat it as an address. These instructions take a type as an immediate argument to know how many bytes to access and which encoding/decoding to use. WebAssembly defines two functions, serialise and deserialise, to encode and decode all four

numerical types. The **load** and **store** instructions can also take additional information (such as an offset) as immediate arguments to allow for simple pointer arithmetic. We show here a simple use of the **load** instruction, where the only immediate argument is the type to read:

$$\frac{F.\mathsf{inst.mems}[0] = k \qquad S.\mathsf{mems}[k][c..c + \mathsf{sizeof}(t)] = bs \qquad \mathsf{deserialise}(t, bs) = c'}{(S, F, [\mathbf{i32.const}\ c;\ t.\mathbf{load}]) \hookrightarrow (S, F, [t.\mathbf{const}\ c'])}$$

Just like for the global variables, the index in the store of the current module's linear memory is looked up in the instance $F.\mathsf{inst}$.[1]

Typing. WebAssembly 1.0 defines a simple type system with only four types: **i32**, **i64**, **f32** and **f64** (as we will see in §2.2, MSWasm introduces a new **handle** type). Instructions have type $t1s \to t2s$, where $t1s$ is the types of the values expected on the stack by the instruction, and $t2s$ is the types of the values that will be pushed on the stack. For example, $t.\mathbf{add}$ has type $[t, t] \to [t]$ and $t.\mathbf{load}$ has type $[\mathbf{i32}] \to [t]$, since addresses into memory are simple **i32** integers in WebAssembly. The WebAssembly type system guarantees that well-typed programs satisfy progress and preservation.

2.2 Segment Memory

In this section, we describe how MSWasm extends WebAssembly with a new kind of memory, *segment memory*, that is accessed not via **i32** integers interpreted as addresses, but via *handles*. More precisely, we present MSWasmCert — our formalisation of MSWasm in Coq — which adapts the prose description of MSWasm to a mechanisation of the full official 1.0 specification, and fixes some minor mistakes and limitations of the original prose definition.

Handles. MSWasm introduces new runtime values, handles, and a corresponding type, **handle**, which is distinct from the numeric types of WebAssembly. A handle is a form of fat pointer, represented as a record with the following fields : a base, an offset, a bound, a valid bit, and an id. The handle points to the bytes beginning at address (base + offset), its bounds of authority is described by the interval [base..base + bound), and its id is used to identify a handle based on its original allocation. Handles are unforgeable, and can only either be derived from other handles, or created when a segment is allocated by the **segalloc** instruction. In particular, this means that **handle**.**const** h cannot occur in the source program, it only appears at runtime. In MSWasmCert, we enforce this syntactically: as shown in Figure 2, handle constants are not *basic instructions*, i.e. instructions available to the programmer, but rather *administrative instructions*, i.e. instructions that only appear at runtime.

Dynamic Checks. A handle does not invariantly require its address base + offset to be within its bounds of authority [base..base + bound), thus allowing for common code patterns where a forbidden pointer might be created but never used (e.g. right before the end of a loop). Instead, instructions that seek to access the segment memory trigger *dynamic checks*, which guarantee that the accessed addresses are within the bounds of authority of the handle, that the validity bit is **true**, and that the handle's id is still live in the *allocator* (see below). If the conditions are met, the **segload** and **segstore** instructions are permitted to read and write from the *segment memory*. If the conditions are not met, the instructions reduce to **trap**. Just like the **load** and **store** instructions in linear memory, the **segload** and **segstore** instructions take a value type as an immediate argument to know how many bytes to read in the segment memory, and how to interpret these bytes.

[1] In WebAssembly 1.0, modules have at most one memory so the list $F.\mathsf{inst.mems}$ is of length at most one, hence the index at which we inspect it is always 0.

Storing Handles in Memory. One subtlety arises from reading handles. If no precautions are taken, a user could write a series of integer values into memory and then read them using **handle**.**segload**, effectively forging a handle. To prevent this, the bytes in segment memory are tagged as either **Handle** or **Numeric**. When reading a handle, if any of the involved bytes is tagged as **Numeric**, the read yields a handle with the validity bit set to **false**. MSWasm also mandates that reading and writing handles can only be done at addresses that are aligned with the length of a handle. This prevents forging a handle, which could otherwise be done by writing two handles consecutively in segment memory and reading from a tagged but unaligned address midway through the first. Since the bytes in linear memory are untagged, reading handles from it would be unsafe as this may forge a handle. Hence using the **load** instuction to read a handle from linear memory automatically traps. If handles must be stored and loaded, this can be done safely in segment memory by using the **segstore** and **segload** instructions.

In MSWasmCert, we abstract over what mechanism is used to serialise a handle into a byte representation, and simply assume we are provided two functions serialise_handle and deserialise_handle.

Operations on Handles. Two new instructions allow for manipulating handles: **handle**.**add** adds to the offset of a handle, changing its address, while **slice** restricts its bounds of authority. Neither operation increases the *authority* of a handle, and thus both are safe. In both cases, the id stays the same, thus uniquely identifying the handle across changes: all handles that share the same id are all derived from one original handle. Accordingly, freeing one handle (see below) will simultaneously free all handles with the same id.

Modules. The original pen-and-paper description of MSWasm [Michael et al. 2023] implicitly assumes that all programs run in the same module, and thus altogether omits any mention of WebAssembly modules (although their implementation reuses rWasm's support for modules). In MSWasmCert, we formally account for the full module system of WebAssembly. To do this, we need to decide how the coarse-grained encapsulation properties of the WebAssembly module system ought to interact with the fine-grained encapsulation properties of MSWasm handles. Rather than operating over several coarsely encapsulated segment memories, we choose to limit the store to a single segment memory shared between all modules. This simplifies the design, and makes it seamless to share a handle from one module to another.[2] This also underlines that the encapsulation properties no longer stem from WebAssembly's module system, but from the handles themselves providing fine-grained memory safety.

Allocator. Handles can be dynamically allocated and freed. While a handle grants spacial authority over the fragment of segment memory described in its metadata, the handle itself does not express whether that region is still temporally valid, or has already been freed. Instead, MSWasm keeps track of live handles using an *allocator*. Michael et al. [2023] state that the allocator should have an 'allocation' and a 'free' function, and describe some of their expected properties. In MSWasmCert, we define the allocator as a map from handle ids to either None, meaning a handle that has been freed, or Some pair of integers representing the handle's original base address and bound. Allocating a handle is modelled by extending the map, and freeing a handle is modelled by updating its mapping to None. The programmer can perform these operations by using the **segalloc** and **segfree** instructions. Allocation is non-deterministic: the handle returned by the **segalloc** instruction could point to any non-live part of segment memory. We impose several (slightly different from Michael et al. [2023]) conditions on the handle to be freed: its base and bound fields must be the original address and bound the handle got allocated as (i.e. the handle cannot have been sliced), its offset must be zero, and its validity bit must be **true**.

[2]Related questions arise in the context of capability machines featuring virtual memory [Watson et al. 2023, §3.11.3].

$$\text{aligned}(a, b) \triangleq a \text{ modulo } b = 0$$
$$\text{compatible}(addr, off, ainst) \triangleq \forall id\ addr'\ off'.\ ainst(id) = \text{Some}(addr', off') \implies (addr + off \le addr' \vee addr \ge addr' + off')$$

$$\frac{addr \le \text{length}(sinst.\text{segdata}) \qquad ainst(id) = \text{None} \qquad \text{compatible}(addr, off, ainst) \qquad sinst' = \{\text{segdata} = sinst.\text{segdata}[addr..addr + off := 0],\ \text{max} = sinst.\text{max}\} \qquad ainst' = ainst[id \mapsto \text{Some}(addr, off)]}{\langle sinst, ainst\rangle \xhookrightarrow{\text{salloc}(addr, off, id)} \langle sinst', ainst'\rangle}$$

We do not require that $addr + off \le \text{length}(sinst.\text{segdata})$, so in cases like $addr = \text{length}(sinst.\text{segdata})$, we are actually growing the segment memory by appending zeros at the end.

$$\frac{ainst(id) = \text{Some}(addr, bound) \qquad ainst' = ainst[id \mapsto \text{None}]}{\langle sinst, ainst\rangle \xhookrightarrow{\text{sfree}(addr, bound, id)} \langle sinst, ainst'\rangle}$$

Fig. 3. Helper functions and helper rules for the allocator

Operational Semantics. The operational semantics rules for MSWasmCert are presented in Figure 4. These rules are almost identical to those of Michael et al. [2023], with the changes brought by MSWasmCert highlighted in indigo. We describe these changes below. For brevity, we do not include failure rules, which dictate that **segload**, **segstore**, **segfree**, **handle.add** and **slice** all reduce to the failure value **trap** if the premises to apply the successful rule are not met. We also provide in Figure 3 our own definitions for the $\langle sinst, ainst\rangle \xhookrightarrow{\text{salloc}(addr, off, id)} \langle sinst', ainst'\rangle$ and $\langle sinst, ainst\rangle \xhookrightarrow{\text{sfree}(addr, bound, id)} \langle sinst', ainst'\rangle$ predicates used in the allocation and freeing rules.

The changes in the reduction rules from MSWasm to MSWasmCert are:

- MSWasmCert introduces a second operational semantics rule for **segalloc**, allowing the allocation to non-deterministically fail, to account for realistic machine behaviour.
- MSWasmCert adds an extra check on the **handle.add** operation to ensure that the new offset is non-negative. This is a design choice that allows us to use unsigned integers to represent offsets, and means that the check for non-negativity of offset in the rules for **segload** and **segstore** are now vacuous in MSWasmCert and can be removed.
- MSWasmCert enforces that freeing a handle must be done with the original handle, not a sliced version of it (the prose definition mandated that the base address must be the original address, but did not enforce that the bound must be the original bound). This is a design choice that we have found convenient when defining our program logic, and it allows for a programmer to easily create a non-freeable handle.
- MSWasmCert fixes two minor typos from the original work [Michael et al. 2023]: the higher bound check in the **segload** and **segstore** rules should be a $\le$ instead of a $<$ (otherwise, when allocating n spaces of memory, one cannot read a value that has size n), and the bound check for the second component of **slice** should be stricter since the bound is an offset from the base rather than an address: changing the base always needs to be compensated by lowering the bound.

$$\frac{t \neq \textbf{handle} \quad \cancel{0 \leq h.\text{offset}} \quad h.\text{offset} + \text{sizeof}(t) \leq h.\text{bound} \quad h.\text{valid} = \textbf{true} \quad \text{isSome}(S.\text{allocator}(h.\text{id})) \quad \mathit{addr} = h.\text{base} + h.\text{offset} \quad S.\text{seg}[\mathit{addr}..\mathit{addr} + \text{sizeof}(t)] = \mathit{tbs} \quad \text{deserialise}(t, \text{untag}(\mathit{tbs})) = c}{(S, F, [\textbf{handle.const}\ h;\ t.\textbf{segload}]) \hookrightarrow (S, F, [t.\textbf{const}\ c])}$$

$$\frac{t = \textbf{handle} \quad \cancel{0 \leq h.\text{offset}} \quad h.\text{offset} + \text{sizeof}(t) \leq h.\text{bound} \quad h.\text{valid} = \textbf{true} \quad \text{isSome}(S.\text{allocator}(h.\text{id})) \quad \mathit{addr} = h.\text{base} + h.\text{offset} \quad S.\text{seg}[\mathit{addr}..\mathit{addr} + \text{sizeof}(t)] = \mathit{tbs} \quad \text{deserialise}(t, \text{untag}(\mathit{tbs})) = h' \quad \text{aligned}(\mathit{addr}, \text{handle\_size}) \quad b = \text{allHandle}(\text{tags}(\mathit{tbs})) \quad h_f = \text{updateValid}(h', b \wedge h'.\text{valid})}{(S, F, [\textbf{handle.const}\ h;\ t.\textbf{segload}]) \hookrightarrow (S, F, [t.\textbf{const}\ h_f])}$$

$$\frac{t \neq \textbf{handle} \quad \cancel{0 \leq h.\text{offset}} \quad h.\text{offset} + \text{sizeof}(t) \leq h.\text{bound} \quad h.\text{valid} = \textbf{true} \quad \text{isSome}(S.\text{allocator}(h.\text{id})) \quad \mathit{addr} = h.\text{base} + h.\text{offset} \quad \text{serialise}(t, c) = \mathit{bs} \quad \mathit{seg}' = S.\text{seg}[\mathit{addr}..\mathit{addr} + \text{sizeof}(t) := \text{addTag}(\mathit{bs}, \textbf{Numeric})] \quad S' = \{S \text{ with seg} = \mathit{sinst}'\}}{(S, F, [\textbf{handle.const}\ h; t.\textbf{const}\ c;\ t.\textbf{segstore}]) \hookrightarrow (S', F, [])}$$

$$\frac{t = \textbf{handle} \quad \cancel{0 \leq h.\text{offset}} \quad h.\text{offset} + \text{sizeof}(t) \leq h.\text{bound} \quad h.\text{valid} = \textbf{true} \quad \text{isSome}(S.\text{allocator}(h.\text{id})) \quad \mathit{addr} = h.\text{base} + h.\text{offset} \quad \text{serialise}(t, h') = \mathit{bs} \quad \mathit{seg}' = S.\text{seg}[\mathit{addr}..\mathit{addr} + \text{sizeof}(t) := \text{addTag}(\mathit{bs}, \textbf{Handle})] \quad S' = \{S \text{ with seg} = \mathit{sinst}'\} \quad \text{aligned}(\mathit{addr}, \text{handle\_size})}{(S, F, [\textbf{handle.const}\ h; \textbf{handle.const}\ h';\ t.\textbf{segstore}]) \hookrightarrow (S', F, [])}$$

$$\frac{\langle S.\text{seg}, S.\text{allocator}\rangle \xrightarrow{\text{salloc}(\mathit{addr}, \mathit{off}, \mathit{id})} \langle \mathit{sinst}', \mathit{ainst}'\rangle \quad S' = \{S \text{ with seg} = \mathit{sinst}',\ \text{allocator} = \mathit{ainst}'\} \quad h = \{\text{base} = \mathit{addr},\ \text{offset} = 0,\ \text{bound} = \mathit{off},\ \text{valid} = \textbf{true},\ \text{id} = \mathit{id}\}}{(S, F, [\textbf{i32.const}\ c;\ \textbf{segalloc}]) \hookrightarrow (S', F, [\textbf{handle.const}\ h])}$$

$$\frac{h = \{\text{base} = 0,\ \text{offset} = 0,\ \text{bound} = 0,\ \text{valid} = \textbf{false},\ \text{id} = 0\}}{(S, F, [\textbf{i32.const}\ c;\ \textbf{segalloc}]) \hookrightarrow (S, F, [\textbf{handle.const}\ h])}$$

$$\frac{\langle S.\text{seg}, S.\text{allocator}\rangle \xrightarrow{\text{sfree}(h.\text{base}, h.\text{bound}, h.\text{id})} \langle \mathit{sinst}', \mathit{ainst}'\rangle \quad S' = \{S \text{ with seg} = \mathit{sinst}',\ \text{allocator} = \mathit{ainst}'\} \quad h.\text{offset} = 0 \quad h.\text{valid} = \textbf{true}}{(S, F, [\textbf{handle.const}\ h;\ \textbf{segfree}]) \hookrightarrow (S', F, [])}$$

$$\frac{h.\text{offset} + c \geq 0 \quad h' = \text{updateOffset}(h, h.\text{offset} + c)}{(S, F, [\textbf{i32.const}\ c;\ \textbf{handle.const}\ h;\ \textbf{handle.add}]) \hookrightarrow (S, F, [\textbf{handle.const}\ h'])}$$

$$\frac{0 \leq c_1 < h.\text{bound} \quad c_1 \leq c_2 \quad h' = \{\text{base} = h.\text{base} + c_1,\ \text{offset} = h.\text{offset},\ \text{bound} = h.\text{bound} - c_2,\ \text{valid} = h.\text{valid},\ \text{id} = h.\text{id}\}}{(S, F, [\textbf{handle.const}\ h;\ \textbf{i32.const}\ c_1;\ \textbf{i32.const}\ c_2;\ \textbf{slice}]) \hookrightarrow (S, F, [\textbf{handle.const}\ h'])}$$

Fig. 4. Operational semantics for the new instructions in MSWasm, phrased in the syntax of MSWasmCert. The non-cosmetic changes brought by MSWasmCert to MSWasm are highlighted in indigo. Clauses made redundant by our mechanisation are crossed out.

Buffer Example. Let us come back to the buffer example from §1.1. We assume function $adv has type [**handle**] → [], but nothing more: it could be imported from another module and we might not know or trust its code. Since we do not share $h with this function, we expect the return value of this program to be 42. In the next section, we present a program logic that lets us prove this.

3 Program Logic

In order to reason about programs written in MSWasm, we define a program logic, Iris-MSWasm. Our program logic allows us to specify and verify known programs, and lays the foundations for defining the logical relation in §4, which allows to reason about interactions with unknown code.

Iris-MSWasm builds on top of Iris-Wasm [Rao et al. 2023], a program logic for WebAssembly, and on the Cerise family [Georges 2023; Georges et al. 2021a, 2022a, 2021b, 2022b; Skorstengaard 2019; Skorstengaard et al. 2018, 2019a,b] of program logics for an idealised capability machine inspired by CHERI. Iris-Wasm captures the coarse-grained encapsulation guarantees of plain WebAssembly, so building on it helps to highlight the differences to the fine-grained encapsulation guarantees we focus on. Building on top of Iris-Wasm also means that we inherit many properties like higher-orderness and the ability to reason about reentrant host calls. While mostly orthogonal to fine-grained memory safety, they can be desirable in many cases.

In this section, we recall Iris-Wasm, and then explain how we adapted it to MSWasm.

3.1 Iris-Wasm

Iris-Wasm [Rao et al. 2023] is a program logic for WebAssembly, defined in the Iris logical framework [Jung et al. 2018]. Instantiated with a language's operational semantics, Iris provides a program logic that allows to prove properties of programs, phrased in a higher-order separation logic. Atop the *structural rules* from Iris, we can derive *instruction-specific proof rules* for each instruction of the language. We can then use them to reason about WebAssembly code in a syntax-directed way.

Logical Values. We define *logical values*, noted w, to describe expressions that cannot reduce. These can be of several kinds. *Immediate values* **immV** *vs* represent a list of WebAssembly values. The *trap value* **trapV** represents a program that has safely halted execution. Iris-Wasm also defines other kinds of logical values because of WebAssembly's expressive control flow mechanisms. The original Iris-Wasm paper describes the treatment of these other logical values, which is unchanged in Iris-MSWasm.

Specifications. We phrase our proof rules and specifications using either *Hoare triples* or *weakest precondition* statements. The Hoare triple $\{P\}\ es\ \{w, \Phi(w)\}$ means that "if the precondition P holds, the expression *es* executes safely while maintaining all invariants, and if it terminates on a logical value w, the predicate Φ holds of that value w". A weakest precondition $\mathrm{wp}\ es\ \{w, \Phi(w)\}$ is a separation logic proposition that means "we hold precisely the resources necessary to run *es* safely and without breaking invariants, and if that run terminates on a logical value w, the predicate Φ holds of that value w".

Resources. The Iris-Wasm program logic defines *resources* that describe ownership of the frame or ownership of fragments of the store; and weakest precondition rules corresponding to each instruction of WebAssembly, dictating what resources are needed to run each instruction. For example, the proof rule for t.**load** is given by (the coloured boxed are used to contrast with our

$i \overset{\text{wm}}{\longmapsto}_{addr} b$	Ownership of a byte in linear memory
$i \overset{\text{wms}}{\longmapsto}_{addr} bv$	Ownership of a list of bytes in linear memory
$\overset{\text{ws}}{\longmapsto}_{addr} tb$	Ownership of a tagged byte in segment memory
$\overset{\text{wss}}{\longmapsto}_{addr} tbs$	Ownership of a list of tagged bytes in segment memory
$id \overset{\text{allocated}}{\longrightarrow}^{q} (addr, bound)^?$	(Fractional) ownership of a handle id in the allocator
$i \overset{\text{wg}}{\longmapsto} \{mutability; v\}$	Ownership of a global variable
$\overset{\text{Fr}}{\hookrightarrow} F$	Ownership of the WebAssembly frame

Fig. 5. Points-to assertions corresponding to various components of the state

wp_segload rule we introduce in §3.2):

wp_load

$$\frac{F.\text{inst.mems}[0] = n \;*\; n \overset{\text{wms}}{\longmapsto}_i bs \;*\; \text{deserialise}(t, bs) = v \;*\; \triangleright\, \Phi(\textbf{immV}\ [v]) \;*\; \overset{\text{Fr}}{\hookrightarrow} F}{\text{wp}\ [\textbf{i32.const}\ i; t.\textbf{load}] \left\{ w,\ \Phi(w) \;*\; \overset{\text{Fr}}{\hookrightarrow} F \;*\; n \overset{\text{wms}}{\longmapsto}_i bs \right\}}$$

Taking $\Phi(w) \triangleq w = \textbf{immV}\ [v]$, this means that if we own the frame resource $\overset{\text{Fr}}{\hookrightarrow} F$ and the linear memory resource[3] $n \overset{\text{wms}}{\longmapsto}_i bs$, the load instruction executes safely. The n on the left-hand-side of the linear memory resource corresponds to the index of this module's memory in the store, looked up in the frame. If the instruction returns (which it does in this case), the return value is v, and we are handed back the frame resource $\overset{\text{Fr}}{\hookrightarrow} F$ and the memory resources $n \overset{\text{wms}}{\longmapsto}_i bs$. Figure 5 displays some of the resources of the Iris-Wasm program logic, with the new resources introduced by Iris-MSWasm highlighted in magenta.

In addition to reasoning about individual WebAssembly modules, Iris-Wasm also introduces a simple host language together with a program logic for it, making it possible to reason about multiple WebAssembly modules being sequentially instantiated by the host environment. The most important piece of this host language program logic is the *instantiation lemma*, that roughly states that if a module typechecks and we own resources corresponding all its imports, the module can be instantiated and we get resources corresponding to all objects (e.g. function closures, linear memories, global variables, etc.) created by the module.

Finally, Iris-Wasm is accompanied by a logical relation that allows to reason about unknown code. We describe our extension of this logical relation in detail in §4.

3.2 Iris-MSWasm

Our program logic, Iris-MSWasm, is defined by adapting Iris-Wasm to the features introduced by MSWasm. This entailed defining the logical ghost state for allocators and segment memories, defining new resources, and proving proof rules for all new instructions.

This constituted a non-trivial programming effort, as many of the new features behave very differently from the existing ones that have been implemented in Iris-Wasm. For example, in plain WebAssembly, all components of the store, including linear memories, grow monotonically during execution, so a simple heap can be used to represent them. But the segment memory can have parts of it freed, so a ghost map had to be used instead of a heap. Additionally, converting a linear memory to an index-map is as simple as mapping all indices from 0 to the size of the memory to

[3]We use superscripts on the arrows (e.g. wms for the linear memory resource) to differentiate the various resources present in the program logic. Some resources like the frame resource additionally use a different arrow shape.

their corresponding byte. For the segment memory, only live addresses should point to a value, increasing the complexity of the definitions.

To accommodate for the new type of memory, we introduce new points-to resources, as described in Figure 5. The segment memory resource $\overset{\text{ws}}{\longmapsto}_{addr}\ tb$ represents ownership of a single tagged byte in segment memory. Allocator resources $id \xrightarrow{\text{allocated}}{}^{q}$ None or $id \xrightarrow{\text{allocated}}{}^{q}$ Some$(addr, bound)$ represent fractional ownership of a handle id in the allocator. q is rational in $(0, 1]$. The case where $q = 1$ represents full ownership and allows to access or modify the value on the right-hand-side of the arrow; in that case we may omit writing the fraction. If $q < 1$, the resource is only partially owned: the right-hand-side value can be accessed, but not modified. Since freeing a handle corresponds to updating its value in the allocator from Some $(base, bound)$ to None, freeing requires full ownership, and we can use partial resources to symbolise handles that are unfreeable because they have been sliced. We also define a syntactic sugar for ownership of a list of tagged bytes tbs in segment memory: $\overset{\text{wss}}{\longmapsto}_{addr} tbs$. Contrary to the resources for linear memory, there is no store index on the left-hand side of the arrow in the segment memory resources. This reflects the fact that all modules share one common segment memory.

Using theses new resources, we define and prove new weakest-precondition rules for all of MSWasm's new instructions. We present these new rules in Figures 6 and 7. These rules mirror the operational semantics introduced in §2.2. For example, the rule for t.**segload** is:

wp_segload

$$\frac{t \neq \textbf{handle} \ast \overset{\text{wss}}{\longmapsto}_{addr} tbs \ast h.\text{id} \xrightarrow{\text{allocated}} \text{Some}(x) \ast h.\text{offset} + \text{sizeof}(t) \leq h.\text{bound} \ast addr = h.\text{base} + h.\text{offset} \ast h.\text{valid} = \textbf{true} \ast \boxed{\text{deserialise}(t, \text{untag}(tbs)) = v \ast \triangleright \Phi(\textbf{immV}\ [v]) \ast \overset{\text{Fr}}{\hookrightarrow} F}}{\text{wp}\ [\textbf{handle.const}\ h; t.\textbf{segload}] \left\{ w, \boxed{\Phi(w) \ast \overset{\text{Fr}}{\hookrightarrow} F} \ast \overset{\text{wss}}{\longmapsto}_{addr} tbs \ast h.\text{id} \xrightarrow{\text{allocated}} \text{Some (x)} \right\}}$$

This rule is quite close to the wp_load rule from §3.1. The differences are (1) the segment rule does dynamic checks to ensure the read is admissible, (2) the memory resource is a linear memory resource in the wp_load rule but a segment memory resource in the wp_segload rule (which also means that the premise looking up an index in the frame instance F.inst is unnecessary in the segment rule), and (3) the allocator resource is additionally present in the segment rule.

The premise $t \neq \textbf{handle}$ in wp_segload is required, because in the case of reading a handle from memory, additional checks are necessary and hence requires a separate wp_segload_handle rule, as displayed in Figure 6. A similar $t \neq \textbf{handle}$ premise has to be added to wp_load in Iris-MSWasm, since reading a handle from linear memory is not allowed in the MSWasm semantics. This is the only modification necessary to a rule from Iris-Wasm when defining Iris-MSWasm.

3.3 Specifying the Known Parts of the Buffer Example

Let us come back to the buffer example from §1.1, whose code is in Figure 1. In this section, we show how to reason about the known parts of its code, and we defer the discussion about the adversary call to §4.3. This explanation is quite technical, because we detail the entire proof. Its mechanisation can be found in our Coq development in file `buffer_code.v`.

Our goal will be to prove that

$$\left\{\overset{\text{Fr}}{\hookrightarrow} F\right\}\ buffer\_example\ \left\{w, (\exists F'.\ \overset{\text{Fr}}{\hookrightarrow} F') \ast (w = \textbf{trapV} \vee w = \textbf{immV}\ [\textbf{i32.const}\ 42])\right\}$$

wp_segload

$$\frac{t \neq \textbf{handle} * \overset{\text{wss}}{\longmapsto}_{addr} tbs * h.\text{id} \xrightarrow{\text{allocated}}^{q} \text{Some}(x) * h.\text{offset} + \text{sizeof}(t) \leq h.\text{bound} * addr = h.\text{base} + h.\text{offset} * h.\text{valid} = \textbf{true} * \text{deserialise}(t, \text{untag}(tbs)) = v * \triangleright \Phi(\textbf{immV}\ [v]) * \overset{\text{Fr}}{\hookrightarrow} F}{\text{wp}\ [\textbf{handle.const}\ h; t.\textbf{segload}]\ \left\{w, \Phi(w) * \overset{\text{wss}}{\longmapsto}_{addr} tbs * h.\text{id} \xrightarrow{\text{allocated}}^{q} \text{Some}(x) * \overset{\text{Fr}}{\hookrightarrow} F\right\}}$$

wp_segload_handle

$$\frac{t = \textbf{handle} * \overset{\text{wss}}{\longmapsto}_{addr} tbs * h.\text{id} \xrightarrow{\text{allocated}}^{q} \text{Some}(x) * h.\text{offset} + \text{sizeof}(t) \leq h.\text{bound} * \text{aligned}(addr, \text{handle\_size}) * b = \text{allHandle}(\text{tags}(tbs)) * h_f = \text{updateValid}(h', b \wedge h'.\text{valid}) * addr = h.\text{base} + h.\text{offset} * h.\text{valid} = \textbf{true} * \text{deserialise}(t, \text{untag}(tbs)) = h' * \triangleright \Phi(\textbf{immV}\ [h_f]) * \overset{\text{Fr}}{\hookrightarrow} F}{\text{wp}\ [\textbf{handle.const}\ h; t.\textbf{segload}]\ \left\{w, \Phi(w) * \overset{\text{wss}}{\longmapsto}_{addr} tbs * h.\text{id} \xrightarrow{\text{allocated}}^{q} \text{Some}(x) * \overset{\text{Fr}}{\hookrightarrow} F\right\}}$$

wp_segload_failure1

$$\frac{\left(h.\text{offset} + \text{sizeof}(t) > h.\text{bound} \vee h.\text{valid} = \textbf{false} \vee (t = \textbf{handle} \wedge \neg\text{aligned}(h.\text{base} + h.\text{offset}, \text{handle\_size}))\right) * \triangleright \Phi(\textbf{trapV}) * \overset{\text{Fr}}{\hookrightarrow} F}{\text{wp}\ [\textbf{handle.const}\ h; t.\textbf{segload}]\ \left\{w, \Phi(w) * \overset{\text{Fr}}{\hookrightarrow} F\right\}}$$

wp_segload_failure2

$$\frac{h.\text{id} \xrightarrow{\text{allocated}}^{q} \text{None} * \triangleright \Phi(\textbf{trapV}) * \overset{\text{Fr}}{\hookrightarrow} F}{\text{wp}\ [\textbf{handle.const}\ h; t.\textbf{segload}]\ \left\{w, \Phi(w) * h.\text{id} \xrightarrow{\text{allocated}} \text{None} * \overset{\text{Fr}}{\hookrightarrow} F\right\}}$$

(We omit the similar failure rules for **segstore**, **segfree**, **handle.add** and **slice**; these rules are shown in our supplementary material)

Fig. 6. Iris-MSWasm rules for the **segload** instruction

where F is a frame where two local variables $h and $hpub are declared, both of type **handle**, and the instance contains a function $adv of type [**handle**] → []. Our desired post-condition allows the program to trap: this could correspond either to the allocation failing, or to the function call failing. Crucially, *trapping is safe*, as it ensures that no memory violation has occurred.

Since we focus here on reasoning about known code, we assume until the end of this section that we know a specification for function $adv, say:

$$\forall h, q.\left\{\begin{array}{l}\overset{\text{wss}}{\longmapsto}_{h.\text{base}} - * \\ h.\text{id} \xrightarrow{\text{allocated}}^{q} \text{Some}-\end{array}\right\}[\textbf{handle.const}\ h; \textbf{call}\ \$adv]\left\{w, \left(\begin{array}{l} w = \textbf{trapV} \vee \\ w = \textbf{immV}\ [] * \\ \exists tbs'.\ \overset{\text{wss}}{\longmapsto}_{h.\text{base}} tbs' * \\ \exists opt.\ h.\text{id} \xrightarrow{\text{allocated}}^{q} opt\end{array}\right)\right\}$$

In other words, the function can be called on any handle input, as long as the caller has ownership of the segment memory region pointed by that handle. The function may trap, but if it does not, it yields back ownership of the same segment memory region upon return; the tagged bytes might

wp_segstore

$$t \neq \textbf{handle} * \overset{\text{wss}}{\longmapsto}_{addr} tbs * h.\text{id} \xrightarrow{\text{allocated}}{}^{q} \text{Some}(x) * h.\text{offset} + \text{sizeof}(t) \leq h.\text{bound} * addr = h.\text{base} + \text{offset} * h.\text{valid} = \textbf{true} * \textbf{typeof}(v) = t * |bs| = \text{sizeof}(t) * \text{serialise}(t, v) = bs * \text{addTag}(bs, \textbf{Numeric}) = tbs' * \triangleright \Phi(\textbf{immV}\ []) * \hookrightarrow^{\text{Fr}} F$$
$$\text{wp}\ [\textbf{handle.const}\ h; v; t.\textbf{segstore}] \left\{ w, \begin{array}{l} \Phi(w) * \overset{\text{wss}}{\longmapsto}_{addr} tbs' * \\ h.\text{id} \xrightarrow{\text{allocated}}{}^{q} \text{Some}(x) * \hookrightarrow^{\text{Fr}} F \end{array} \right\}$$

wp_segstore_handle

$$t = \textbf{handle} * \overset{\text{wss}}{\longmapsto}_{addr} tbs * h.\text{id} \xrightarrow{\text{allocated}}{}^{q} \text{Some}(x) * h.\text{offset} + \text{sizeof}(t) \leq h.\text{bound} * addr = h.\text{base} + \text{offset} * h.\text{valid} = \textbf{true} * \text{aligned}(addr, \text{handle\_size}) * |bs| = \text{sizeof}(t) * \text{serialise}(t, h') = bs * \text{addTag}(bs, \textbf{Handle}) = tbs' * \triangleright \Phi(\textbf{immV}\ []) * \hookrightarrow^{\text{Fr}} F$$
$$\text{wp}\ [\textbf{handle.const}\ h; \textbf{handle.const}\ h'; t.\textbf{segstore}] \left\{ w, \begin{array}{l} \Phi(w) * \overset{\text{wss}}{\longmapsto}_{addr} tbs' * \\ h.\text{id} \xrightarrow{\text{allocated}}{}^{q} \text{Some}(x) * \hookrightarrow^{\text{Fr}} F \end{array} \right\}$$

wp_segfree

$$h.\text{valid} = \textbf{true} * h.\text{offset} = 0 * |tbs = b| * \overset{\text{wss}}{\longmapsto}_{h.\text{base}} tbs * h.\text{id} \xrightarrow{\text{allocated}} \text{Some}(h.\text{base}, b) * \triangleright \Phi(\textbf{immV}\ []) * \hookrightarrow^{\text{Fr}} F$$
$$\text{wp}\ [\textbf{handle.const}\ h; \textbf{segfree}] \left\{ w, \Phi(w) * \hookrightarrow^{\text{Fr}} F \right\}$$

wp_segalloc

$$\triangleright \left(\forall w. \left(\exists h. w = \textbf{immV}\ [\textbf{handle.const}\ h] * (h.\text{valid} = \textbf{false} \vee (\text{id} \xrightarrow{\text{allocated}} \text{Some}(h.\text{base}, n) * h.\text{bound} = n * h.\text{offset} = 0 * h.\text{valid} = \textbf{true} * \overset{\text{wss}}{\longmapsto}_{h.\text{base}} \text{repeat}(n, 0)))) \mathrel{-\!\!*} \Phi(w) \right) * \hookrightarrow^{\text{Fr}} F$$
$$\text{wp}\ [\textbf{i32.const}\ n; \textbf{segalloc}] \left\{ w, \Phi(w) * \hookrightarrow^{\text{Fr}} F \right\}$$

wp_handleadd

$$h' = \{\text{base} = h.\text{base},\ \text{offset} = h.\text{offset} + c,\ \text{bound} = h.\text{bound},\ \text{valid} = h.\text{valid},\ \text{id} = h.\text{id}\} * h.\text{offset} + c \geq 0 * \triangleright \Phi(\textbf{immV}\ [\textbf{handle.const}\ h']) * \hookrightarrow^{\text{Fr}} F$$
$$\text{wp}\ [\textbf{i32.const}\ c;\ \textbf{handle.const}\ h;\ \textbf{handle.add}] \left\{ w, \Phi(w) * \hookrightarrow^{\text{Fr}} F \right\}$$

wp_slice

$$h' = \{\text{base} = h.\text{base} + c_1,\ \text{offset} = h.\text{offset},\ \text{bound} = h.\text{bound} - c_2,\ \text{valid} = h.\text{valid},\ \text{id} = h.\text{id}\} * 0 \leq c_1 < h.\text{bound} * c_1 \leq c_2 * \triangleright \Phi(\textbf{immV}\ [\textbf{handle.const}\ h']) * \hookrightarrow^{\text{Fr}} F$$
$$\text{wp}\ [\textbf{handle.const}\ h;\ \textbf{i32.const}\ c_1;\ \textbf{i32.const}\ c_2;\ \textbf{slice}] \left\{ w, \Phi(w) * \hookrightarrow^{\text{Fr}} F \right\}$$

Fig. 7. Other Iris-MSWasm specific rules

have changed. We will return in §4.3 to the more general case where the function is completely untrusted and we are not given a specification for it.

Let us proceed instruction by instruction, recalling the resources we own at each point of the program. The resources displayed in gold are the ones necessary to fulfil premises of the next rule to be applied; the ones in black are unused and simply carried forward.

$$\{\hookrightarrow^{\text{Fr}} F\}$$

Lines 0–1. At the start, we only own the frame resource $\hookrightarrow^{\text{Fr}} F$. We can apply[4] rule wp_segalloc from Figure 7 with

$$\Phi(w) \triangleq \big(\exists h.w = \textbf{immV}\ [\textbf{handle.const}\ h] * (h.\text{valid} = \textbf{false} \lor (\text{id} \xrightarrow{\text{allocated}} \text{Some}(h.\text{base}, n) * h.\text{bound} = n * h.\text{offset} = 0 * h.\text{valid} = \textbf{true} * \mapsto^{\text{wss}}_{h.\text{base}} \text{repeat}(n, 0))))$$

(hence the wand implication in the first premise is a trivial $P \mathrel{-\!\!*} P$). To satisfy the second premise, we yield the resource $\hookrightarrow^{\text{Fr}} F$. The post-condition gives us back the resource $\hookrightarrow^{\text{Fr}} F$, and tells us that a value **handle.const** h has now been placed on the stack, and that either $h.\text{valid} = \textbf{false}$ (representing a failed allocation), or we own the segment and allocator resources. If we define $x \triangleq (h.\text{base}, h.\text{bound})$, we have:

$$\big\{\hookrightarrow^{\text{Fr}} F * \big(h.\text{valid} = \textbf{false} \lor \mapsto^{\text{wss}}_{h.\text{base}} \text{repeat}(8, 0) * h.\text{id} \xrightarrow{\text{allocated}} \text{Some}\ x\big)\big\}$$

Lines 2–3. The next two instructions are **local.set** and **local.get**. Both instructions have corresponding proof rules in Iris-Wasm, which we apply sequentially. In both cases, the Iris-Wasm proof rule consumes the frame resource $\hookrightarrow^{\text{Fr}} F$ as a premise, and gives it back in the post-condition. **local.set** changes the frame to new one, F', where the value of local variable $h is now h:

$$\big\{\hookrightarrow^{\text{Fr}} F' * \big(h.\text{valid} = \textbf{false} \lor \mapsto^{\text{wss}}_{h.\text{base}} \text{repeat}(4 + 4, 0) * h.\text{id} \xrightarrow{\text{allocated}} \text{Some}\ x\big)\big\}$$

Lines 4–5. The next instruction is **segstore**. At this stage, we perform a case disjunction: if $h.\text{valid} = \textbf{false}$ (i.e. the allocation has failed), then the failure rule wp_segstore_failure1 (the **segstore** equivalent of rule wp_segload_failure1 from Figure 6) applies since one of the dynamic checks fails. Hence we trap safely, and in this case we can conclude the whole proof here, as we have filled the first disjunct of the post-condition.

Let us now consider the second case: we own $\mapsto^{\text{wss}}_{h.\text{base}} \text{repeat}(8, 0)$. We can apply rule wp_segstore from Figure 7 with $\Phi(w) \triangleq w = \textbf{immV}\ []$. To fulfil the segment resource premise of the rule, we must yield the first half of the resource we hold. Thus we separate $\mapsto^{\text{wss}}_{h.\text{base}} \text{repeat}(8, 0)$ into two resources $\mapsto^{\text{wss}}_{h.\text{base}} \text{repeat}(4, 0)$ and $\mapsto^{\text{wss}}_{h.\text{base}+4} \text{repeat}(4, 0)$. We yield the first of these (as well as the frame resource $\hookrightarrow^{\text{Fr}} F'$ and our allocator resource) to satisfy the premises of wp_segstore, and the latter is unused for this rule. The other premises are all the necessary dynamic checks, which are satisfied here, and the rules give us our resources back, having updated the tagged bytes in segment memory to now store our private value 42.

$$\big\{\hookrightarrow^{\text{Fr}} F' * \mapsto^{\text{wss}}_{h.\text{base}} \text{serialise}(\textbf{i32}, 42) * \mapsto^{\text{wss}}_{h.\text{base}+4} \text{repeat}(4, 0) * h.\text{id} \xrightarrow{\text{allocated}} \text{Some}\ x\big\}$$

Lines 6–11. The next instructions are **local.get**, **slice**, **local.set** and **local.get** again. All of these instructions have associated proof rules: wp_slice from Figure 7 for **slice**, and rules from Iris-Wasm for the local variables. The rule for **local.set** has changed the frame again to update the value of variable $hpub to h', the "second half" of h that we obtained via slicing; we call F'' this new frame.

$$\big\{\hookrightarrow^{\text{Fr}} F'' * \mapsto^{\text{wss}}_{h.\text{base}} \text{serialise}(\textbf{i32}, 42) * \mapsto^{\text{wss}}_{h.\text{base}+4} \text{repeat}(4, 0) * h.\text{id} \xrightarrow{\text{allocated}}{}^{1/2+1/2} \text{Some}\ x\big\}$$

[4]We omit the structural rules that allow to bind the first instruction in order to apply the proof rule.

Line 12. Now, we come to the call to function $adv. In our simplified setting, we have a specification, which we wish to apply. Since by definition $h'.\mathsf{base} = h.\mathsf{base} + 4$ and $h'.\mathsf{id} = h.\mathsf{id}$, we have all the resources needed to fill the precondition. If we apply the specification with $q = 1$, we must lose the entire allocator resource to fulfil the precondition of the specification, and we would only get back that there exists *opt* such that $h.\mathsf{id} \xrightarrow{\text{allocated}} opt$. This would not allow us to later execute the **segload** instruction. Instead, we can separate our allocator resource into two partial resources $h.\mathsf{id} \xrightarrow{\text{allocated}}^{1/2} \mathsf{Some}\ x$. Now we can apply the specification with $q = \frac{1}{2}$ yielding only one of our partial resources, and keeping the second.

The postcondition tells us that either the call has trapped (in which case we can terminate the proof like before), or there exists some tagged bytes tbs' and an option *opt* such that we now own $\overset{\text{wss}}{\longmapsto}_{h'.\mathsf{base}} tbs'$ and $h.\mathsf{id} \xrightarrow{\text{allocated}}^{1/2} opt$. Combined with the partial resource we kept, we know that $opt = \mathsf{Some}\ x$, and we can combine our two fragments to get a full allocator resource. Informally, that means that the handle is still allocated.

Importantly, the other handle is not required by the specification, and hence the segment resource $\overset{\text{wss}}{\longmapsto}_{h.\mathsf{base}} \mathsf{serialise}(\mathbf{i32}, 42)$ is framed away.

$$\left\{\hookrightarrow^{\text{Fr}} F'' * \overset{\text{wss}}{\longmapsto}_{h.\mathsf{base}} \mathsf{serialise}(\mathbf{i32}, 42) * \overset{\text{wss}}{\longmapsto}_{h.\mathsf{base}+4} tbs' * h.\mathsf{id} \xrightarrow{\text{allocated}} \mathsf{Some}\ x\right\}$$

Lines 13–14. Lastly, we use the Iris-Wasm rule for **local.get** to get the value of variable $h, and rule wp_segload from Figure 6 allows us to conclude that the return value is indeed 42 as expected.

In the next section, we show how we can achieve the same result when the function $adv is not specified.

3.4 Adequacy

The Iris logical framework provides an *adequacy theorem* [Jung et al. 2018, §6.4] that relates the weakest precondition statement to the operational semantics. This means that Iris is not in our Trusted Computing Base, as holding a weakest precondition now implies a statement phrased entirely in terms of the operational semantics of MSWasmCert.

Theorem 3.1 (Adequacy). *If* $\mathsf{wp}\ es\ \{w, \Phi(w)\}$ *and* $(S, F, es) \hookrightarrow^* (S', F', vs)$ *for some values vs, then* $\Phi(vs)$ *holds.*

Using the adequacy theorem, we can prove the following result for the buffer example from §1.1:

Theorem 3.2 (Buffer Example). *If the code in Figure 1 terminates, it terminates on either the trap value* **trapV***, or on value 42*

Proof sketch. We begin by proving

$$\left\{\hookrightarrow^{\text{Fr}} F\right\} \mathtt{buffer\_example} \left\{w, (\exists F'.\ \hookrightarrow^{\text{Fr}} F') * (w = \mathbf{trapV} \vee w = \mathbf{immV}\ [\mathbf{i32.const}\ 42])\right\}$$

We have shown in §3.3 how to reason about the known parts of the code, and we will show in §4.3 how to reason about the unknown code; hence we have the wanted Hoare triple. This proof can also be seen in our Coq development in file `buffer_code.v`.

Then, we use the program logic for our host language to reason about the instantiation on the adversary module and the buffer module. The instantiation lemma provides the frame resource $\hookrightarrow^{\text{Fr}} F$ from the precondition of the Hoare triple. This yields the weakest precondition statement $\mathsf{wp}\ \mathtt{buffer\_instantiation}\ \{w, w = \mathbf{trapV} \vee w = \mathbf{immV}\ [\mathbf{i32.const}\ 42]\}$. A proof of this can be seen in our Coq development in file `buffer_instantiation`.

Finally, we apply the adequacy theorem which yields the desired result. This entails carefully providing all of the resource algebras necessary to implement the logical state of Iris and use all

the ghost resources that Iris-MSWasm leverages. A mechanised proof can be seen in our Coq development in file `buffer_adequacy.v`. □

4 Robust Capability Safety

We have described how to use Iris-MSWasm to reason about known code. What remains to verify a complete example is to explain how to reason about unknown, potentially adversarial code. More precisely, when proving the weakest precondition for the buffer example, we eventually reach the call to the unknown imported function. At that point, one of our proof obligation is to show the weakest precondition for the body of that function. Since the function is arbitrary, we cannot step through its instructions. And since the function is untrusted, we cannot simply assume that we are given a weakest precondition for it. Instead, we want to define a *universal* specification for unknown code, which gives an over-approximation of its behaviour in the form of a weakest precondition.

To that end, we define a logical relation for the MSWasm type system, and prove that it satisfies the fundamental theorem of logical relations. In essence, the logical relation defines what it means for a value to be *safe to share*, and an expression to be *safe to execute*. Our logical relation builds on the logical relation defined in Iris-Wasm [Rao et al. 2023], and follows the typical design of step-indexed logical relations [Ahmed 2004] in Iris [Krebbers et al. 2017; Timany et al. 2022], and applies the techniques used in the Cerise line of work [Georges et al. 2021a, 2022a,b]. We present the intuition behind our logical relation in §4.1, and then define it and show that it is sound in §4.2, and showcase how it gives us robust safety on our buffer example in §4.3.

4.1 Informal Intuition

The high-level idea behind our logical relation is to define what it means for a value to be *safe to share*, and an expression to be *safe to execute*. What this means depends on the type of the value or expression: for example, a handle is safe to share if it grants memory access to its range of authority (i.e. grants access to the relevant points-to predicates), and if that memory recursively contains values that are safe to share. Meanwhile, an expression *es* is safe to execute when there is a weakest precondition for it $\mathsf{wp}\ es\ \{w, w \text{ is safe to share}\}$. In this simplified definition, *es* either loops, or reduces to a value that is safe to share. The formal definition has to account for programs that reduce to **trapV**, as well as programs that either **return** or **break** to the surrounding context. Crucially, as described earlier, a program that reduces to **trapV** (say, because it failed a dynamic check) is safe to execute.[5]

The definitions of *safe to share* and *safe to execute* can be viewed as a *universal contract*, in the sense that it holds for all well-typed MSWasm programs. A key theorem is to prove that this is the case. We call this result the *fundamental theorem of the logical relation*: if a program *es* is a well-typed MSWasm program, then it is *safe to execute*. We state this theorem formally in §4.2.

By applying the fundamental theorem, since module instantiation guarantees that its functions are well typed, we can derive weakest precondition specifications for imported functions, even when they are unknown. The caveat is that in order to get this specification, any shared handle must *also* satisfy the universal contract, i.e. satisfy the value interpretation for handles. Thus, a key feature of the logical relation is to capture the fine-grained encapsulation properties of handles, so as not to impose invariants over segment regions that are *not* shared.

[5] As mentionned in §1.2, we only consider integrity properties. If we were to consider confidentiality properties, we would need to consider potential interoperability with IO

$$\boxed{\mathcal{V}[\![ts]\!] : \mathit{LogVal} \to \mathit{iProp}}$$

$$\mathrm{ValidHandleAddr}(addr, base', bound') \triangleq \mathrm{aligned}(base' + addr, \mathrm{handle\_size}) \wedge 0 \le addr \wedge addr + \mathrm{handle\_size} \le bound'$$

$$\mathcal{V}_0[\![\mathbf{handle}]\!](v) \triangleq \exists h.\ v = \mathbf{handle.const}\ h * \left(\begin{array}{l} h.\mathrm{valid} = \mathbf{false} \ \vee \\ \exists \gamma, base', bound', base'', bound'', q. \\ [h.\mathrm{base}..h.\mathrm{base} + h.\mathrm{bound}) \subseteq [base'..base' + bound') * \quad (1) \\ [base'..base' + bound') \subseteq [base''..base'' + bound'') * \quad (2) \\ q \in \left\{\frac{1}{2}, 1\right\} * \left((h.\mathrm{base} = base'' * h.\mathrm{bound} = bound'') \implies q = 1\right) * \quad (3) \\ \boxed{\circ\,(h.\mathrm{id} \hookrightarrow (\gamma, base'', bound'', q))}^{\gamma_{\mathrm{toks}}} * \quad (4) \\ \boxed{\begin{array}{l} \exists tbs. |tbs| = bound' * \overset{\mathrm{wss}}{\longmapsto}_{base'} tbs * \\ \quad \forall addr.\ \mathrm{ValidHandleAddr}(addr, base', bound') \mathrel{-\!\!*} \\ \quad \left(\begin{array}{l} \left(\begin{array}{l} \exists off.\ 0 \le off < \mathrm{handle\_size} \wedge \\ \quad tbs[addr + off] = (-, \mathbf{Numeric}) \end{array} \right) \vee \\ \mathcal{V}_0[\![\mathbf{handle}]\!](\mathbf{handle.const} \\ \quad \mathrm{deserialise\_handle}(\\ \quad\quad \mathrm{untag}(tbs[addr..addr + \mathrm{handle\_size}])) \end{array} \right) \end{array}}^{\mathrm{CInv}} \quad (5) \\ \end{array} \right)^{\gamma}$$

$$\mathcal{V}_0[\![t]\!](v) \triangleq \exists c.\ v = t.\mathbf{const}\ c \qquad (\text{for } t \neq \mathbf{handle})$$

$$\mathcal{V}[\![[t_1, \cdots, t_n]]\!](w) \triangleq w = \mathbf{trapV} \vee \exists v_1, \cdots, v_n.\ w = \mathbf{immV}\ [v_1, \cdots, v_n] \wedge \mathcal{V}_0[\![t_1]\!](v_1) \wedge \cdots \wedge \mathcal{V}_0[\![t_n]\!](v_n)$$

Fig. 8. Our logical relation for values

4.2 Logical Relation

More formally, we define, for each type t, a predicate $\mathcal{V}[\![t]\!]$ describing values that are safe to share, called the *value interpretation* of type t, and a predicate $\mathcal{E}[\![t]\!]$ of expressions that are safe to execute, called the *expression interpretation* of type t.

The difficulty when defining a logical relation for a full industrial language is that one must define a logical interpretation for all objects of the language: not only values and expressions, but also frames, function closures, linear memories, instances, contexts, etc. Iris-Wasm defines a relation for each WebAssembly object. In this work, we extend it to interpret the new types introduced by MSWasm. In particular, we define new interpretations for handle values and allocators. To keep the explanations simple, we will primarily focus on these new logical relations, and refer to the Coq mechanisation for the full definition. That being said, the explanations are somewhat technical, and will assume some familiarity with various Iris concepts.

Value Interpretation. The value interpretation is shown in Figure 8. It states that a logical value is safe for types *ts* if it either is the trap value **trapV** (recall that we consider expressions that trap to be safe), or if it is a list of WebAssembly values which satisfy $\mathcal{V}_0$ for each value type in *ts*.

$\mathcal{V}_0$ defines the interpretation of value types, namely numerical types and handles. For numerical types, Iris-Wasm simply asserts that the numerical value has the appropriate format (32 bit integer for **i32**, etc.). It is interesting to note, that although **i32**s are used to access linear memory, $\mathcal{V}_0$ does not model this usage. Indeed, this is by design: although **i32**s are used as pointers, it is the instance within the frame that provides the *authority* to access linear memory. As such, it is the interpretation

$$\boxed{\mathcal{A} : \mathit{Allocator} \to \mathit{iProp}}$$

$$\begin{array}{rcll} \mathrm{cinvOpt}(\mathit{base}, \mathit{bound}, y) & \triangleq & \begin{cases} \mathit{base} = b' * \mathit{bound} = e' * [\mathrm{CInv} : \gamma] & \mathit{if}\ y = \mathrm{Some}(b', e') \\ \top & \mathit{otherwise} \end{cases} & \\ \mathcal{A}(\mathit{allctr}) & \triangleq & \exists f. \boxed{\bullet f}^{\gamma_{\mathrm{tok}}} * \ast_{(\mathit{id} \mapsto (\gamma, \mathit{base}, \mathit{bound}, q)) \in f} & \exists y.\ \mathit{allctr}(\mathit{id}) = y * \\ & & & \mathit{id} \xrightarrow{\text{allocated}}{}^{q}\ y * \\ & & & \mathrm{cinvOpt}(\mathit{base}, \mathit{bound}, y) \end{array}$$

Fig. 9. Our logical relation for the allocator

of linear memory that determines which points-to predicates can be used by a function. In short, the interpretation of linear memory imposes an invariant over the *entirety* of a module's linear memory, thus expressing how a module may forge pointers to access any byte within it.

Meanwhile, handles specifically do *not* grant authority over the entire segment memory. Instead, our goal is to model the exact authority granted by a handle: namely the authority to access the locations within its bounds of authority, and the authority to free a handle if its bounds match the original bounds of that handle as represented in the allocator.

Let us take a more detailed look at our definition for the value interpretation for handles in Figure 8. It states that a value is in the interpretation for handles if it is a handle h, and either this handle is invalid, or we own (1) a range $[\mathit{base}'..\mathit{base}' + \mathit{bound}')$, representing the bounds of the handle when it was originally *shared*[6], (2) a range $[\mathit{base}''..\mathit{base}'' + \mathit{bound}'')$, representing the bounds of the handle when it was originally *allocated* (we want to remember so we can determine whether the handle grants the authority to be freed[7]) (3) a fraction q that can be $\frac{1}{2}$ or 1, and has to be 1 if h.base = base'' and h.bound = bound'' (this fraction will be used to model the authority to free a handle) , (4) a ghost resource binding h.id to an invariant name γ, the range $[\mathit{base}''..\mathit{base}'' + \mathit{bound}'')$ and the fraction q (this resource will be used to remember the original state of a handle, at its allocation), and (5) an invariant that contains the segment memory locations associated to the range $[\mathit{base}'..\mathit{base}' + \mathit{bound}')$, such that all locations in memory that might store a handle (i.e. are in bounds and are aligned that are aligned with the handle size) either have at least one byte tagged as **Numeric**, or hold a value that satisfies the value interpretation for handles.

The ghost resource (4) is a fragment view of a map whose authoritative view is in the interpretation for the allocator. In other words, this ghost resource serves to share information about handle ids between the value interpretation and the allocator interpretation, as we will detail later. The ghost name γ_{toks} is a global value that is also used by the interpretation for the allocator.

Since handles can be freed, we use Iris' *cancellable invariants* [Jung et al. 2018, §7.1.3]. A cancellable invariant uses a token $[\mathrm{CInv} : \gamma]$ to track whether an invariant is still live. This token is required to open the invariant, and can be consumed to cancel the invariant when the handle is being freed, after which the segment memory resources become unavailable. Previous mechanisations of robust capability safety [Georges et al. 2022a; Swasey et al. 2017] do not consider temporal safety properties of heap memory. Most existing mechanisations also make the simplifying assumption that memory locations hold full objects rather than individual bytes like in MSWasm. Alignment concerns are responsible for part of the complexity of our definition.

[6]This bound may be greater than the current bounds — recall that handle slicing makes it safe to share any of its sub-bounds.
[7]The handle that was originally allocated might have strictly larger bounds than the handle that was originally shared, as is the case in the running buffer example.

Allocator Intepretation. Let us discuss the interpretation for the allocator, shown in Figure 9. It asserts that there exists a mapping f from handle ids to invariant names, such that this mapping agrees with the fragments from the handle value interpretation, and for every binding in this map, there is a corresponding binding in the allocator and a corresponding *partial* allocator resource. If that binding is to a live handle, then we additionally require that we hold the token that will allow us to open the cancellable invariant from the handle value interpretation.

In other words, the handle value interpretation holds the spatial resources inside a cancellable invariant, and the allocator resource holds the token that allows to open said invariant as long as the handle is live, thereby maintaining the temporal authority. The former expresses persistent knowledge over segment memory, while the latter expresses non-duplicable knowledge of the allocator.

The allocator resource is partial with degree q, meaning it can only be modified if $q = 1$, else it can only be inspected but not updated. This means that code looking to free a handle (i.e. update the resource from Some($base$, $bound$) to None) must own $q = 1$.

Allocator and Handle Interpretation Together. Let us assume we own the interpretation for a handle value h together with the interpretation for an allocator, and see how we can reason about running **segload**, **segstore** or **segfree** on h.

First, we proceed by cases on h.valid: if it is false, then all three instructions trap safely. Else, we now hold two ghost resources: one from the value interpretation of the handle, and one from the interpretation for the allocator. Combining them yields a binding $h.\mathsf{id} \mapsto \gamma, base'', bound'', q$ for which the allocator interpretation gives us a corresponding binding in the allocator, as well as an allocator resource $h.\mathsf{id} \xrightarrow{\text{allocated}}{}^{q} y$. We can then perform a case distinction on y: if it is None then the handle has been freed and all three instructions will safely trap; else, the allocator interpretation gives a token that can be used to open the invariant in the value interpretation for the handle.

Once the invariant is open, we hold both the segment resources for the area of memory pointed by h, and an allocator resource for h.id. This is enough to perform a read or a write on h. In that case, y has remained unchanged, so the invariant can be trivially closed again, giving back the token for the interpretation for the allocator.

In the last case, if the instruction is a **segfree**, we must proceed by case on whether h.base and h.bound are equal to the values present in the allocator (which the cinvOpt in the allocator interpretation tells us are equal to $base''$ and $bound''$). If they aren't, the freeing operation safely traps. If they are, we can cancel the invariant instead of closing it. The value interpretation mandates that $q = 1$ and hence we can update the allocator resource to $h.\mathsf{id} \xrightarrow{\text{allocated}}$ None, which we can use to restore the interpretation for the allocator without needing the cancellable invariant token.

As illustrated above, all the necessary resources are obtainable when holding the allocator interpretation and the value interpretation for handles. Hence we do not need an interpretation for the full segment memory, unlike for linear memory. This reflects the fine-grained reasoning that segment memory allows: memory is never considered in its entirety, but only handle by handle.

Expression Interpretation. Figure 10 shows excerpts from the definition of the Iris-Wasm logical relation, with the few modifications brought by Iris-MSWasm in magenta. These modifications are the addition of the allocator and corresponding allocator interpretation. We use a weakest precondition to define that an expression is safe to run. During the execution of a WebAssembly expression, the expression might not terminate on a WebAssembly value, but rather on a **br** or **return** instruction, or on a host call; hence the four-way disjunction in the postcondition. We give definitions for $\mathcal{H}$, $\mathcal{B}r$, and $\mathcal{R}et$ in our supplementary material. The post-condition also yields back frame and allocator resources. This expression interpretation is most interesting when considered

$$\boxed{\mathit{Frame}[\![ts]\!]_{inst} : \mathit{Frame} \to \mathit{iProp}}$$

$$\mathit{Frame}[\![ts]\!]_{inst}(F) \triangleq [\mathrm{NaInv} : \top] * \xhookrightarrow{\mathrm{Fr}} F * \exists vs.\, F = \{inst; vs\} * \mathcal{V}[\![ts]\!](\mathbf{immV}\ vs)$$

$$\boxed{\mathcal{E}[\![ts]\!]^{*}_{*} : \mathit{Expr} \to \mathit{iProp}}$$

$$\mathcal{E}[\![ts]\!]^{\tau_{\mathrm{lbs}},\tau_{\mathrm{ret}}}_{(\tau l,inst,hfs)}(lh, es) \triangleq \mathrm{wp}\ es \left\{ w,\ \begin{array}{l} \left(\begin{array}{l} \mathcal{V}[\![ts]\!](w) \vee \mathcal{H}[\![ts]\!]^{\tau_{\mathrm{lbs}},\tau_{\mathrm{ret}}}_{(\tau l,inst,hfs)}(w) \vee \\ \mathcal{B}r[\![\tau_{\mathrm{lbs}}]\!]^{\tau_{\mathrm{ret}}}_{(\tau l,inst,hfs)}(w, lh) \vee \mathcal{R}et[\![\tau_{\mathrm{ret}}]\!]_{(\tau l,inst)}(w) \end{array} \right) * \\ \exists F, allctr.\, \mathit{Frame}[\![\tau l]\!]_{inst}(F) * \mathcal{A}(allctr) \end{array} \right\}$$

$$\boxed{C \vDash es\colon ts1 \to ts2}$$

$$C \vDash es\colon ts1 \to ts2 \triangleq \forall inst, lh, hfs.\, (\mathcal{I}[\![C]\!](inst) * \mathit{Ctx}[\![C]\!]_{(inst,hfs)}(lh)) \mathrel{-\!\!*} \forall F, allctr, vs.\, (\mathcal{V}[\![ts1]\!](vs) * \xhookrightarrow{\mathrm{Fr}} F * \mathit{Frame}[\![\tau l]\!]_{inst}(F) * \mathcal{A}(allctr)) \mathrel{-\!\!*} \mathcal{E}[\![ts2]\!]^{\tau_{\mathrm{lbs}},\tau_{\mathrm{ret}}}_{(\tau l,inst,hfs)}(lh, vs +\!\!+ es)$$

where $\tau l = C.\mathrm{locals}$, $\tau_{\mathrm{lbs}} = C.\mathrm{labels}$, and $\tau_{\mathrm{ret}} = C.\mathrm{return}$.

Fig. 10. Excerpts from the definition of our logical relation

together with the definition of *semantic typing*, also given in Figure 10. An expression is semantically well typed (written with a double turnstile $\vDash$ instead of the simple turnstile $\vdash$ used for syntactic typing) when, given a context and instance that are safe to use (see our supplementary material for definitions of $\mathcal{I}$ and Ctx), as well as arguments that are safe to share, a frame, and an allocator, the resulting expression is in the expression interpretation.

We can now state the fundamental theorem of the logical relation:

Theorem 4.1 (Fundamental theorem of the logical relation). *If a program bs (a list of* basic instructions*, i.e. only using instructions available to the programmer) typechecks syntactically, then it typechecks semantically:*

$$\forall bs, C, ts1, ts2.\, C \vdash bs\colon ts1 \to ts2 \implies C \vDash bs\colon ts1 \to ts2$$

Proof sketch. The proof proceeds by induction on the syntactic typing judgement. The added challenge with respect to its prior version is to prove the cases for the new segment instructions, each of which depend on the new value relation for handles, and the interpretation of the allocator. A full proof can be found in the Coq development. □

4.3 Robust Safety

Buffer Example. Let us come back to our buffer example from Figure 1 and show how we can reason about the call to the unknown, untrusted function $adv. All we assume is that this function is well typed in MSWasm's typing system, with type $[\mathbf{handle}] \to []$.

Jumping back into the proof detailed in §3.3, right before the call, we own the following resources:

$$\left\{ \xhookrightarrow{\mathrm{Fr}} F'' * \overset{\mathrm{wss}}{\longmapsto}_{h.\mathrm{base}} \mathrm{serialise}(\mathbf{i32}, 42) * \overset{\mathrm{wss}}{\longmapsto}_{h.\mathrm{base+4}} \mathrm{repeat}(4, 0) * h.\mathrm{id} \xrightarrow{\mathrm{allocated}} \mathrm{Some}\ x \right\}$$

Using the second segment memory resource, we can instantiate an invariant and allocate a ghost resource, giving us $h' \in \mathcal{V}[\![\mathbf{handle}]\!]$. To get $\mathcal{A}(\{h.\mathrm{id} \mapsto \mathrm{Some}(h.\mathrm{base}, h.\mathrm{bound})\})$, we need to also give an allocator resource. Just like in §3.3, we can separate our allocator resource into two fragment resources $h.\mathrm{id} \xrightarrow{\mathrm{allocated}}{}^{1/2} \mathrm{Some}\ x$, and only give one of these to get the $\mathcal{A}$ statement; this allows us to keep partial ownership which lets us know that the handle cannot have been freed.

Hence we can apply the fundamental theorem, and know that our call executes safely, and terminates on a value that is safe to share for type [], i.e. the trap value or the unit value. We also get that there exists a new allocator *allctr′* such that $\mathcal{A}(allctr')$. The segment resource $\overset{\text{wss}}{\longmapsto}_{h.\text{base}}\text{serialise}(\mathbf{i32}, 42)$ was unused and hence the caller has held on to them, and we can use these to complete the proof of the specification.

Robust Safety. This approach, where we leverage the fundamental theorem of the logical relation to prove specifications in the presence of unknown code, allows us to call library functions from untrusted libraries safely, establishing invariants of the form "No matter what untrusted module calls the functions I export, my internal state will satisfy this invariant". This showcases the strength of MSWasm and the fine-grained safety properties it brings to WebAssembly.

5 Stack Example

In this section, we illustrate how our program logic and logical relation scale to a bigger example: a library implementing stacks of **i32** integers. This library builds on a case study from Iris-Wasm [Rao et al. 2023], but uses handles to enforce stronger guarantees. Since plain WebAssembly does not have handles, the stack library of Rao et al. [2023] uses **i32** integers to represent stacks. These values are forgeable, hence the stack library must be encapsulated from untrusted code to prevent the corruption of allocated stacks — technically, by instantiating the adversary without access to the functions of the stack library. In MSWasm, we use handles instead of **i32** integers to represent stacks. With handles, the adversary cannot corrupt the stacks even when it has access to the stack library — technically, when it is instantiated after the stack library, with access to its functions — and we prove this using our logical relation.

Our stack module defines a function $new_stack which uses the **segalloc** instruction to allocate one page (64KiB) of segment memory. Handles to this stack point to the start of this page, and range over all of it. In the first four bytes of the allocated region, we store the *stack pointer* as an offset to the top of the stack, initially the **i32** integer 0. When accessing a stack, we get the offset by loading from the handle, and then combine the handle with the offset by **handle.add** to address the top of the stack.

From here, it is straightforward to define the usual stack operations like $push, $pop, $stack_length, $is_full, and $is_empty. In addition, we define $stack_map, which takes as arguments a stack and a function (more precisely, an index in a table of functions) that it maps on all the elements of the stack. This map function is interesting, because when the function it maps is an adversary function, the execution context has authority over the stack.

In our Coq development, we verify the stack module, and exercise it on key scenarios using different client modules. Here, we focus on a specific client module, $RobustModule in Figure 11, to showcase robust capability safety. This module creates a stack, pushes two values onto it, maps an adversary function (imported from an untrusted *adversary module*) onto the stack, and then asks for the stack's length. We wish to prove that mapping the adversary function does not affect the stack's length. Figure 11 shows the sequence of instantiations performed by the host code.

This example is interesting because both the adversary module and $RobustModule have access to the stack functions and can thus interact with the stack module's memory. Importantly, the adversary function will only be given *values* fetched from the stack module's memory by the $stack_map function, and not handles. If it had access to a handle, the adversary might be able to, say, pop elements from $RobustModule's stack, and then the final length of the stack would change. This form of attack is made impossible by the fact that the handles that represent stacks are unforgeable. Because $RobustModule never shares its handle with the adversary module, the adversary module cannot push, pop, or perform any operations on *that* stack; it may only use

the stack module to create its own stacks and perform operations on those. This showcases the strength MSWasm adds to plain WebAssembly.

We prove the following theorem:

Theorem 5.1 (Robust stack example). *If the host code h_{code} from Figure 11 terminates, it terminates on either the trap value* **trapV***, or on the* **i32** *value 2.*

In particular, this means the adversary cannot push or pop on the client module's stack. A full proof can be found in our Coq development, and we give a succinct overview here.

Proof sketch. In essence, the proof is similar to the one for the buffer example from §3.4. However, since there are multiple modules involved and since some resources must be placed in invariants to apply the fundamental theorem of the logical relation, the order in which the steps on the proof is carried out is crucial.

We begin by proving specifications for all the functions of the stack module. Since this is known code, the proofs are similar to that of §3.3.

We then apply the instantiation lemma three times in a row. First we instantiate the stack module, which does not make any imports, and hence we do not need any resources; the lemma gives us resources corresponding to each individual function closure. Then we instantiate the adversary module. The instantiation lemma requires function closure resources for the stack functions, which we have, and gives these resources back as well as an extra function closure resource corresponding to the adversary function $advf. Finally, we instantiate $RobustModule. Again, the instantiation lemma requires function closure resources for the stack functions as well as the $advf function, and gives these resources back together with an extra function closure resource corresponding to the main function of $RobustModule.

All that remains to do is run the code of $RobustModule. Like in the buffer example, we need to apply the fundamental theorem in order to reason about the unknown function $advf. One additional subtlety we face here is that because the adversary module imports the stack functions, we must first show that these functions are safe to share. These functions are defined in the stack module, hence we must prove all components of that module to be safe. This actually includes the adversary function $advf itself, since that function was placed in the stack module's function table when instantiating $RobustModule. Since these functions can call each other, there is a circularity, which we address (as is standard) by Löb induction. To do this, we need to allocate invariants corresponding to all objects that will need to be proved safe to share. The induction then gives us that the adversary function $advf is safe to share, and in particular we have a weakest precondition that we can use to reason about calls to it.

Using this, we can specify the code of $RobustModule like we did for the buffer example in §3.4 and obtain a weakest precondition. Finally, we apply the adequacy theorem from §3.4 to get the desired result.

□

6 Discussion and Related Works

We discuss prior work that we build on (§6.1), and then return to the question of sharing state (§6.2)

6.1 Prior Work

Iris-Wasm. MSWasmCert is a conservative extension of WasmCert, and Iris-MSWasm is accordingly an extension of Iris-Wasm. In particular, we inherit all the separation logic proof rules for the constructs of WebAssembly, and add new proof rules for the new constructs of MSWasm. Our

$RobustModule:

```
Create a stack s
Push 42 and 10 onto s
Map $advf onto s
Set x := length(s)
Return x
```

Host code h_{code}:

Instantiate $stackmodule
    No imports
    Export *stack functions*
Instantiate $advmodule
    Import *stack functions*
    Export $advf
Instantiate $RobustModule
    Import *stack functions*
    Import $advf
    No exports

Fig. 11. Pseudo-code for the robust stack client, and host code for the instantiation sequence

logical relation is correspondingly an extension of the logical relation of Iris-Wasm: in the absence of handles and segment memory, it collapses to the logical relation of Iris-Wasm.

Cerise. Iris-Wasm uses the same ideas as Cerise, but in the setting of WebAssembly rather than that of capability machines. The main differences are that: the MSWasm allocator enforces temporal safety; MSWasm only feature a single type of permission, which corresponds to an 'RW' write-and-read permission (considering other types of permissions would be interesting); and MSWasm functions and their local arguments are handled by the language, not implemented using capabilities, so MSWasm does not feature function-flavoured capabilities (executable permission, stack-local capabilities, etc.). However, the most significant difference is not one of feature, but of scale: as illustrated in §4.2, our logical relation needs to cover all the language constructs of MSWasm, including frames, the module system, etc., and these pose a significant challenge.

MSWasm's Memory Safety. When introducing MSWasm, Michael et al. [2023] propose a new formal, colour-based definition of memory safety, that captures spatial and temporal memory safety, and pointer integrity. Their definition hinges on a monitor that inspects the execution trace of the program, and checks that the memory accesses agree with the colouring. Concretely, the monitor maintains a *shadow memory* that associates every address with its allocation state (either Allocated or Free), its colour (an arbitrary identifier), and its shade (for intra-object safety). Using the shadow memory as reference, the monitor then checks every event in the trace, making sure that (1) any read and write events are performed on allocated addresses, using the right colour and shade, (2) allocation events only allocate free addresses, and (3) freeing events only free allocated addresses, for which a corresponding allocation event exists in the trace, and no free event since. A trace is said to be *memory safe* if the monitor does not get stuck. While this approach allows them to capture a notion of memory safety, it does not directly make it possible to reason about the combination of known and unknown, potentially adversarial code. First, the monitor does not distinguish events emitted by trusted modules, and events emitted by untrusted ones. In other words, the monitor does not have any notion of private and public state. In the example of §1.1, the monitor does not know whether the read(h) event comes from the known code, or if it comes from the adversary: the monitor accepts the trace in both cases, as the event is legal according to the shadow memory. Second, the monitor definition does not keep track of the values read and written by the memory events. This is especially limiting for reasoning about functional properties, and for keeping track of how values are preserved throughout adversary calls. In the example of §1.1, the colour-based monitor does not check whether the value stored and read by the handle h is 42.

In this paper, we are interested in fine-grained interactions between trusted, known code and untrusted, unknown, arbitrary code. Those properties usually require keeping track of the values preserved throughout adversary calls, by carefully over-approximating the behaviour of the adversary. Adapting the monitor-based definition to tractably bound the set of traces that the adversary can generate would require addressing the frame problem [McCarthy and Hayes 1981].

Our notion of *capability safety* is built on top of a separation logic, and takes account of ownership of resources. It offers an explicit distinction between private and public state: values shared with unknown code need to be owned by the logical relation. As explained in section §4.2, the logical relation recursively computes the addresses reachable from a given value. Giving ownership of an address over to the logical relation gives away knowledge of its contents, as its value is now existentially quantified. Crucially, the frame rule of separation logic keeps track of the private state during an adversary call: the private value are simply framed away.

Michael et al. [2023] use their monitor-based definition of memory safety in the context of secure compilation, which we do not explore in this paper.

6.2 Sharing State in WebAssembly

The rigid nature of WebAssembly 1.0 means that C cannot be compiled in the 'naive' way to WebAssembly. For example, C local variables are too expressive to be compiled to WebAssembly locals, and therefore most production C-to-WebAssembly compilers compile the C stack to a data structure in linear memory. Lehmann et al. [2020] illustrate how this and other limitations mean that many isolation mechanisms provided by usual OS infrastructure for process hardening are not available when compiling to WebAssembly.

As described in the introduction, capabilities are one approach to address this. However, they raise some challenges: common compiler optimisations violate capability safety [Zaliva et al. 2024], and so writing optimising capability-safety-preserving compilers is an open problem; and capability compression on hardware causes a mismatch between source and target languages. This is particularly problematic for MSWasm, as it is meant as an intermediate language, both compiled to, and compiled from. Nonetheless, by making MSWasm precise and proving that it satisfies robust capability safety, we ensure that projects exploring its use as an intermediate language can rely on its design being validated, and can know exactly what MSWasm guarantees.

RichWasm [Paraskevopoulou et al. 2024] extends WebAssembly with a static notion of capability, at the type level instead of at runtime, and use them to statically enforce safe fine-grained sharing.

As a closely related approach, WebAssembly is being enriched with aggregate types [Rossberg 2024] that WebAssembly 2.0 references can point to. WebAssembly references are opaque and unforgeable, and as such act as a simple form of capabilities. Developing a logical relation that captures both handles and references would make it possible to make a more formal comparison.

A very different approach, taken by the WebAssembly Component Model [The Bytecode Alliance 2023a,b] and adopted by several vendors of WebAssembly for cloud computing is to eschew sharing in favour of copying. One of the aims of the WebAssembly Component Model is to allow language interoperability, which typically requires marshalling, and so already incurs the cost of copying.

Our work lays the foundation to evaluate the language-level guarantees of these different approaches, and we hope that it informs future developments and deployments.

6.3 Semantic Language Integrity

Maintaining datatype and notation consistency in a large language specification is challenging, especially if one wants to be able to automatically extract human-readable rules, an interpreter for testing, and theorem prover definitions [Mulligan et al. 2014; Owens et al. 2011; Sewell et al. 2007]. WebAssembly is now getting a DSL [Breitner et al. 2023] for that specific purpose.

However, maintaining the semantic integrity of the language is as important as maintaining its syntactic integrity. The key properties that form the *universal contract* of the language make it possible for specifiers, implementers, and users to work together instead of against each other. In this paper, we demonstrated how to capture key aspect of such a universal contract for as complex an extension of WebAssembly as MSWasm.

Acknowledgments

This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation, for Birkedal, and by an AUFF Starter Grant for Pichon-Pharabod. This work was co-funded by the European Union (ERC, CHORDS, 101096090). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

Data Availability Statement

The artifact [Legoupil et al. 2024] of this paper, containing the full Coq development, is available on Zenodo. Detailed instructions of usage are provided within the artifact itself. The appendix, which contains the full version of figures that had to be shortened in the paper, can be found together with the artifact.

The code is also available on github at https://github.com/logsem/MSWasm.

References

Amal Jamil Ahmed. 2004. *Semantics of types for mutable state.* Ph. D. Dissertation. Princeton University.

Thomas Bauereiss, Brian Campbell, Thomas Sewell, Alasdair Armstrong, Lawrence Esswood, Ian Stark, Graeme Barnes, Robert N. M. Watson, and Peter Sewell. 2022. Verified Security for the Morello Capability-enhanced Prototype Arm Architecture. In *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13240)*, Ilya Sergey (Ed.). Springer, 174–203. https://doi.org/10.1007/978-3-030-99336-8_7

Joachim Breitner, Philippa Gardner, Jaehyun Lee, Sam Lindley, Matija Pretnar, Xiaojia Rao, Andreas Rossberg, Sukyoung Ryu, Wonho Shin, Conrad Watt, and Dongjun Youn. 2023. Wasm SpecTec: Engineering a Formal Language Standard. *CoRR* abs/2311.07223 (2023). https://doi.org/10.48550/ARXIV.2311.07223 arXiv:2311.07223

Matt Butcher. 2022. How to Think About WebAssembly (Amid the Hype). https://www.fermyon.com/blog/how-to-think-about-wasm

Lin Clark. 2019. Announcing the Bytecode Alliance: Building a secure by default, composable future for WebAssembly. https://hacks.mozilla.org/2019/11/announcing-the-bytecode-alliance/

Jack B. Dennis and Earl C. Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9, 3 (mar 1966), 143–155. https://doi.org/10.1145/365230.365252

Dominique Devriese, Lars Birkedal, and Frank Piessens. 2016. Reasoning about Object Capabilities with Logical Relations and Effect Parametricity. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*. IEEE, 147–162. https://doi.org/10.1109/EUROSP.2016.22

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, HASP@ISCA 2019, June 23, 2019*. ACM, 4:1–4:8. https://doi.org/10.1145/3337167.3337171

Fastly documentation. 2022. Compute@Edge. https://docs.fastly.com/products/compute-at-edge

Aïna Linn Georges. 2023. *Designing and Proving Robust Safety of Efficient Capability Machine Programs.* Ph. D. Dissertation. Aarhus University.

Aïna Linn Georges, Armaël Guéneau, Thomas Van Strydonck, Amin Timany, Alix Trieu, Sander Huyghebaert, Dominique Devriese, and Lars Birkedal. 2021a. Efficient and provable local capability revocation using uninitialized capabilities. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434287

Aïna Linn Georges, Armaël Guéneau, Thomas van Strydonck, Amin Timany, Alix Trieu, Dominique Devriese, and Lars Birkedal. 2022a. *Cerise: Program Verification on a Capability Machine in the Presence of Untrusted Code.* Technical Report. Aarhus University. https://cs.au.dk/~birke/papers/cerise.pdf

Aïna Linn Georges, Armaël Guéneau, Thomas Van-Strydonck, Amin Timany, Dominique Trieu, Alix Devriese, and Lars Birkedal. 2021b. Cap' ou pas cap' ?: Preuve de programmes pour une machine à capacités en présence de code inconnu. In *Journées Francophones des Langages Applicatifs 2021*. https://cris.vub.be/ws/portalfiles/portal/55081793/paper.pdf

Aïna Linn Georges, Alix Trieu, and Lars Birkedal. 2022b. *Le Temps des Cerises: Efficient Temporal Stack Safety on Capability Machines using Directed Capabilities*. Technical Report. Aarhus University. https://cs.au.dk/~ageorges/publications_pdfs/monotone-technical.pdf

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 185–200. https://doi.org/10.1145/3062341.3062363

Pat Hickey. 2020. How Fastly and the developer community are investing in the WebAssembly ecosystem. https://www.fastly.com/blog/how-fastly-and-developer-community-invest-in-webassembly-ecosystem

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. https://doi.org/10.1017/S0956796818000151

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive proofs in higher-order concurrent separation logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 205–217. https://doi.org/10.1145/3009837.3009855

Maxime Legoupil, June Rousseau, Aïna Linn Georges, Jean Pichon-Pharabod, and Lars Birkedal. 2024. *Artifact and Appendix of 'Iris-MSWasm: elucidating and mechanising the security invariants of Memory- Safe WebAssembly'*. https://doi.org/10.5281/zenodo.13383121

Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 217–234. https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

J. McCarthy and P.J. Hayes. 1981. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Readings in Artificial Intelligence*, Bonnie Lynn Webber and Nils J. Nilsson (Eds.). Morgan Kaufmann, 431–450. https://doi.org/10.1016/B978-0-934613-03-3.50033-7

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. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 1–15. https://doi.org/10.1145/2908080.2908081

Alexandra E. Michael, Anitha Gollamudi, Jay Bosamiya, Evan Johnson, Aidan Denlinger, Craig Disselkoen, Conrad Watt, Bryan Parno, Marco Patrignani, Marco Vassena, and Deian Stefan. 2023. MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code. *Proc. ACM Program. Lang.* 7, POPL (2023), 425–454. https://doi.org/10.1145/3571208

Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 175–188. https://doi.org/10.1145/2628136.2628143

Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony C. J. Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. 2020. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1003–1020. https://doi.org/10.1109/SP40000.2020.00055

Scott Owens, Peter Böhm, Francesco Zappa Nardelli, and Peter Sewell. 2011. Lem: A Lightweight Tool for Heavyweight Semantics. In *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6898)*, Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk (Eds.). Springer, 363–369. https://doi.org/10.1007/978-3-642-22863-6_27

Zoe Paraskevopoulou, Michael Fitzgibbons, Noble Mushtak, Michelle Thalakottur, Jose Sulaiman Manzur, and Amal Ahmed. 2024. *RichWasm: Bringing Safe, Fine-Grained, Shared-Memory Interoperability Down to WebAssembly*. Technical Report. arXiv:2401.08287 https://arxiv.org/pdf/2401.08287.pdf

Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1096–1120. https://doi.org/10.1145/3591265

Andreas Rossberg. 2019. *WebAssembly Core Specification W3C Recommendation*. Technical Report. W3C. https://www.w3.org/TR/wasm-core-1/

Andreas Rossberg. 2024. *WebAssembly Specification Release 2.0 + tail calls + function references + gc (Draft 2024-03-19).* Technical Report. https://webassembly.github.io/gc/core/syntax/types.html

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Tom Ridge, Susmit Sarkar, and Rok Strnisa. 2007. Ott: effective tool support for the working semanticist. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, Ralf Hinze and Norman Ramsey (Eds.). ACM, 1–12. https://doi.org/10.1145/1291151.1291155

Lau Skorstengaard. 2019. *Formal Reasoning about Capability Machines.* Ph. D. Dissertation. Aarhus University.

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2018. Reasoning About a Machine with Local Capabilities - Provably Safe Stack and Return Pointer Management. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings.* 475–501. https://doi.org/10.1007/978-3-319-89884-1_17

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019a. Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management. *ACM Transactions on Programming Languages and Systems* 42, 1 (Dec. 2019), 5:1–5:53. https://doi.org/10.1145/3363519

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019b. StkTokens: Enforcing Well-Bracketed Control Flow and Stack Encapsulation Using Linear Capabilities. *Proc. ACM Program. Lang.* 3, POPL, Article 19 (Jan. 2019), 28 pages. https://doi.org/10.1145/3290332

David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and compositional verification of object capability patterns. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 89:1–89:26. https://doi.org/10.1145/3133913

The Bytecode Alliance. 2023a. Component Model design and specification (GitHub repository). https://github.com/WebAssembly/component-model

The Bytecode Alliance. 2023b. The WebAssembly Component Model. https://component-model.bytecodealliance.org/

Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. 2022. A Logical Approach to Type Soundness. (2022). https://cs.au.dk/~timany/publications/files/2022-submitted-logical-type-soundness.pdf

Thomas Van Strydonck, Frank Piessens, and Dominique Devriese. 2019. Linear capabilities for fully abstract compilation of separation-logic-verified code. *Proc. ACM Program. Lang.* 3, ICFP, Article 84 (jul 2019), 29 pages. https://doi.org/10.1145/3341688

Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Hesham Almatary, Jonathan Anderson, John Baldwin, Graeme Barnes, David Chisnall, Jessica Clarke, Brooks Davis, Lee Eisen, Nathaniel Wesley Filardo, Franz A. Fuchs, Richard Grisenthwaite, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, Simon W. Moore, Steven J. Murdoch, Kyndylan Nienhuis, Robert Norton, Alexander Richardson, Peter Rugg, Peter Sewell, Stacey Son, and Hongyan Xia. 2023. *Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 9).* Technical Report UCAM-CL-TR-987. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-987

Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. 2021. Two Mechanisations of WebAssembly 1.0. In *Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13047)*, Marieke Huisman, Corina S. Pasareanu, and Naijun Zhan (Eds.). Springer, 61–79. https://doi.org/10.1007/978-3-030-90870-6_4

M. V. Wilkes and R. M. Needham. 1979. *The Cambridge CAP Computer and Its Operating System.* Elsevier. https://www.microsoft.com/en-us/research/publication/the-cambridge-cap-computer-and-its-operating-system/

Jonathan Woodruff, Paul Metzger, Robert N. M. Watson, Brooks Davis, Wes Filardo, Jessica Clarke, and John Baldwin. 2023. SOSP 2023 CHERI Exercises. https://www.cl.cam.ac.uk/~pffm2/sosp2023_cheri_tutorial/cover/README.html

Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014.* IEEE Computer Society, 457–468. https://doi.org/10.1109/ISCA.2014.6853201

Vadim Zaliva, Kayvan Memarian, Ricardo Almeida, Jessica Clarke, Brooks Davis, Alex Richardson, David Chisnall, Brian Campbell, Ian Stark, Robert N. M. Watson, and Peter Sewell. 2024. Formal Mechanised Semantics of CHERI C: Capabilities, Provenance, and Undefined Behaviour. http://www.cl.cam.ac.uk/users/pes20/asplos24spring-paper110.pdf

Received 2024-04-05; accepted 2024-08-18