Iris-MSWasm: elucidating and mechanising the security invariants of Memory-Safe WebAssembly

Maxime Legoupil

Aarhus Universitet

June 3, 2024

WebAssembly

- Widely used in industry
- Full formal specification
- Code organised in modules
- Users care about encapsulation

How can we show formally the encapsulation guarantees between modules?

Rossberg et al. 2018, *Bringing the web up to speed with WebAssembly*

WebAssembly

- Widely used in industry
- Full formal specification
- Code organised in modules
- Users care about encapsulation

How can we show formally the encapsulation guarantees between modules? Iris-Wasm

---

Rossberg et al. 2018, *Bringing the web up to speed with WebAssembly*

Stack Module

Stack module

Client module

- S defines library functions
- C imports functions from S
- Some form of memory sharing required
- Encapsulation crucial: C' should not get to mess up the stacks C allocated!

Can we give specifications to these modules that ensure encapsulation?

Rao, Georges, **Legoupil**, Watt, Pichon-Pharabod, Gardner, Birkedal. 2023, *Iris-Wasm: Robust and Modular Verification of WebAssembly Programs*

Stack Module

Specification for S:

There exists a predicate **isStack**, such that:

$$\begin{array}{rll}
\{\} & [\textbf{call } \$\mathrm{new\_stack}] & \{v, \textbf{isStack}(v, [])\} \\
\{\textbf{isStack}(v_0, x :: s)\} & [v_0; \textbf{call } \$\mathrm{pop}] & \{v, v = x * \textbf{isStack}(v_0, s)\} \\
\{\textbf{isStack}(v_0, s)\} & [x; v_0; \textbf{call } \$\mathrm{push}] & \{\_, \textbf{isStack}(v_0, x :: s)\} \\
 & \vdots &
\end{array}$$

Rao, Georges, **Legoupil**, Watt, Pichon-Pharabod, Gardner, Birkedal. 2023, *Iris-Wasm: Robust and Modular Verification of WebAssembly Programs*

Stack Module

Stack module

Client module

- We have a specification for S
- We can write a specification for C
- Those can be combined modularly

What if we don't trust the clients?

Rao, Georges, **Legoupil**, Watt, Pichon-Pharabod, Gardner, Birkedal. 2023, *Iris-Wasm: Robust and Modular Verification of WebAssembly Programs*

Stack Module

Stack module

Client module

- We have a specification for S
- We can write a specification for C
- Those can be combined modularly

What if we don't trust the clients? Use robust safety

Rao, Georges, **Legoupil**, Watt, Pichon-Pharabod, Gardner, Birkedal. 2023, *Iris-Wasm: Robust and Modular Verification of WebAssembly Programs*

Robust Safety

- Reason about unknown, untrusted code
- Specifying a module with library calls
- Establishing invariants for a library

Rao, Georges, **Legoupil**, Watt, Pichon-Pharabod, Gardner, Birkedal. 2023, *Iris-Wasm: Robust and Modular Verification of WebAssembly Programs*

Robust Safety

- Reason about unknown, untrusted code
- Specifying a module with library calls
- Establishing invariants for a library

Stack module

Unknown module

- We can still prove some invariants
- Memory layout cannot be changed
- Malicious client can still push and pop

Rao, Georges, **Legoupil**, Watt, Pichon-Pharabod, Gardner, Birkedal. 2023, *Iris-Wasm: Robust and Modular Verification of WebAssembly Programs*

Coarse-grained safety

> *At worst, a buggy or exploited WebAssembly program can make a mess of the data in its own memory*
>
> Rossberg et al. 2018, *Bringing the web up to speed with WebAssembly*

Coarse-grained safety

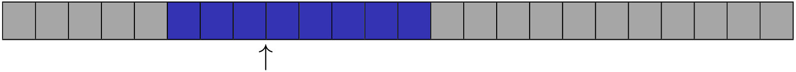
> *At worst, a buggy or exploited WebAssembly program can make a mess of the data in its own memory*
>
> Rossberg et al. 2018, *Bringing the web up to speed with WebAssembly*

What is a useful module's own memory?
E.g. compilers reimplement a call stack in memory!

Lehmann et al. 2020, *Everything Old is New Again: Binary Security of WebAssembly*

Obtaining finer-grained safety

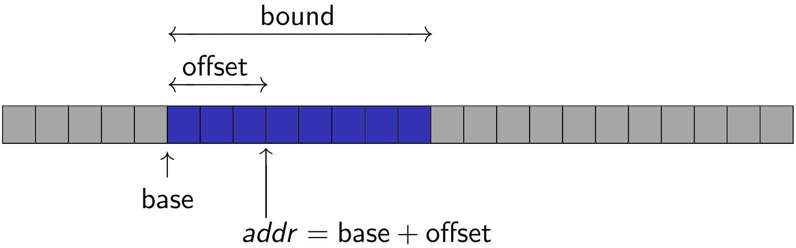
Using a pointer



Using a capability

MSWasm

$$\text{(handles) } h \quad ::= \quad \{\text{base}, \quad \text{offset}, \quad \text{bound}, \quad \text{valid}, \quad \text{id}\}$$



bound

offset

base

$addr = \text{base} + \text{offset}$

Michael et al. 2023, *MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code*

Reading with a handle

$$\text{(handles) } h \quad ::= \quad \{\text{base}, \quad \text{offset}, \quad \text{bound}, \quad \text{valid}, \quad \text{id}\}$$

$$\frac{h.\text{offset} \leq h.\text{bound} \qquad h.\text{valid} = \textbf{true} \qquad \mathrm{isAllocated}(h.\text{id}) \qquad S.\text{seg}[h.\text{base} + h.\text{offset}] = c}{[h;\ t.\textbf{segload}] \hookrightarrow [c]}$$

Note: $h.\text{offset} \geq 0$ is guaranteed by invariant

Michael et al. 2023, *MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code*

Weaknesses of the original proposal

- Pen-and-paper
- Missing some technical details
- Definition of memory safety is unintuitive and difficult to use in practice

How can we make the MSWasm proposal more precise and useful?

Michael et al. 2023, *MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code*

Weaknesses of the original proposal

- Pen-and-paper
- Missing some technical details
- Definition of memory safety is unintuitive and difficult to use in practice

How can we make the MSWasm proposal more precise and useful?

Iris-MSWasm

Michael et al. 2023, *MSWasm: Soundly Enforcing Memory-Safe Execution of Unsafe Code*

A minimalist example

$\{g \xmapsto{\text{wg}} -\}$

```
Allocate two handles $h$ and $h'$
Write 42 to $h$
Call an adversary function $[\textbf{handle}] \to []$ with arg $h'$
Read from $h$
Set global variable $g$ to the read value
Free $h$
```

$\{v, v = \textbf{trap} \vee v = () * g \xmapsto{\text{wg}} 42\}$

Robust Safety by Logical Relation

We define $\Gamma \vDash prog : t_1 \rightarrow t_2$ meaning "*prog* is safe to execute"

Goal: fundamental theorem

$\Gamma \vdash prog : t_1 \rightarrow t_2 \twoheadrightarrow \Gamma \vDash prog : t_1 \rightarrow t_2$

$$\Gamma \vDash prog : t_1 \rightarrow t_2 \triangleq \forall v, v \in \mathcal{V}[\![t_1]\!] \twoheadrightarrow \mathrm{wp}\ v \mathbin{+\!\!+} prog\ \{w, w \in \mathcal{V}[\![t_2]\!]\}$$

$$\mathcal{V}[\![t]\!] \triangleq \{\text{values of type } t \text{ that are safe to use}\}$$

Robust Safety by Logical Relation

We define $\Gamma \vDash prog : t_1 \to t_2$ meaning "*prog* is safe to execute"

Goal: fundamental theorem

$\Gamma \vdash prog : t_1 \to t_2 \mathrel{-\!\!*} \Gamma \vDash prog : t_1 \to t_2$

$$\Gamma \vDash prog : t_1 \to t_2 \triangleq \forall v, v \in \mathcal{V}[\![t_1]\!] \mathrel{-\!\!*} \mathrm{wp}\ v \mathbin{+\!\!+} prog\ \{w, w \in \mathcal{V}[\![t_2]\!]\}$$

$$\begin{aligned}
\mathcal{V}_0[\![\mathrm{Int}]\!] &\triangleq \text{all integers} \\
\mathcal{V}_0[\![\textbf{handle}]\!] &\triangleq \{h \text{ such that } \neg h.\mathsf{valid}, \text{ or} \\
&\qquad \text{we own } \xmapsto{\mathsf{wss}}_{h.\mathsf{base}} bs \text{ and} \\
&\qquad \text{any handle stored there is in } \mathcal{V}_0[\![\textbf{handle}]\!]\} \\
\mathcal{V}[\![t]\!] &\triangleq \{\textbf{trap}\} \cup \mathcal{V}_0[\![t]\!]
\end{aligned}$$

Robust example

```
$\{g \xmapsto{\text{wg}} -\}$
Allocate two handles $h$ and $h'$
Write 42 to $h$
Call an adversary function $[\textbf{handle}] \to []$ with arg $h'$
Read from $h$
Set global variable $g$ to the read value
Free $h$
$\{v, v = () * g \xmapsto{\text{wg}} (0 \text{ or } 42)\}$
```

Robust example

$\{g \xmapsto{\text{wg}} -\}$

`Allocate two handles` h `and` h'

$\{g \xmapsto{\text{wg}} - * (\neg h.\text{valid} \vee \xmapsto{\text{wss}}_{h.\text{base}} 0) * (\neg h'.\text{valid} \vee \xmapsto{\text{wss}}_{h'.\text{base}} 0)\}$

`wp_segalloc_simplified`

$$\frac{}{\text{wp}\,[n; \mathbf{segalloc}]\,\left\{v, v = h * (\neg h.\text{valid} \vee \xmapsto{\text{wss}}_{h.\text{base}} 0)\right\}}$$

Robust example

$\{g \xmapsto{\mathsf{wg}} -\}$

`Allocate two handles` h `and` h'

$\{g \xmapsto{\mathsf{wg}} - * (\neg h.\mathsf{valid} \vee \xmapsto{\mathsf{wss}}_{h.\mathsf{base}} 0) * (\neg h'.\mathsf{valid} \vee \xmapsto{\mathsf{wss}}_{h'.\mathsf{base}} 0)\}$

`Write 42 to` h

$\{g \xmapsto{\mathsf{wg}} - * (\mathbf{trap} \vee \qquad\qquad\qquad\qquad) * (\neg h'.\mathsf{valid} \vee \xmapsto{\mathsf{wss}}_{h'.\mathsf{base}} 0)\}$

`wp_segstore_failure_simplified`

$$\frac{\neg h.\mathsf{valid}}{\mathsf{wp}\ [h; v_0; t.\mathbf{segstore}]\ \{w, w = \mathbf{trap}\}}$$

Robust example

$\{g \xmapsto{\mathsf{wg}} -\}$

`Allocate two handles` h `and` h'

$\{g \xmapsto{\mathsf{wg}} - * (\neg h.\mathsf{valid} \vee \xmapsto{\mathsf{wss}}_{h.\mathsf{base}} 0) * (\neg h'.\mathsf{valid} \vee \xmapsto{\mathsf{wss}}_{h'.\mathsf{base}} 0)\}$

`Write 42 to` h

$\{g \xmapsto{\mathsf{wg}} - * (\mathbf{trap} \vee \xmapsto{\mathsf{wss}}_{h.\mathsf{base}} 42) * (\neg h'.\mathsf{valid} \vee \xmapsto{\mathsf{wss}}_{h'.\mathsf{base}} 0)\}$

`wp_segstore_simplified`

$$\frac{t \neq \mathbf{handle} * \xmapsto{\mathsf{wss}}_{h.\mathsf{base}+h.\mathsf{offset}} - -\!* \ \text{dynamic checks}}{\mathsf{wp}\ [h; v_0; t.\mathbf{segstore}]\ \left\{\xmapsto{\mathsf{wss}}_{h.\mathsf{base}+h.\mathsf{offset}} v_0\right\}}$$

Robust example

$\{g \xmapsto{\text{wg}} -\}$

`Allocate two handles` h `and` h'

$\{g \xmapsto{\text{wg}} - * (\neg h.\mathsf{valid} \vee \xmapsto{\text{wss}}_{h.\mathsf{base}} 0) * (\neg h'.\mathsf{valid} \vee \xmapsto{\text{wss}}_{h'.\mathsf{base}} 0)\}$

`Write 42 to` h

$\{g \xmapsto{\text{wg}} - * (\mathbf{trap} \vee \xmapsto{\text{wss}}_{h.\mathsf{base}} 42) * (\neg h'.\mathsf{valid} \vee \xmapsto{\text{wss}}_{h'.\mathsf{base}} 0)\}$

`Call an adversary function` $[\mathbf{handle}] \to []$ `with arg` h'

Robust example

$\{g \xmapsto{\text{wg}} -\}$

`Allocate two handles` h `and` h'

$\{g \xmapsto{\text{wg}} - * (\neg h.\text{valid} \vee \xmapsto{\text{wss}}_{h.\text{base}} 0) * (\neg h'.\text{valid} \vee \xmapsto{\text{wss}}_{h'.\text{base}} 0)\}$

`Write 42 to` h

$\{g \xmapsto{\text{wg}} - * (\textbf{trap} \vee \xmapsto{\text{wss}}_{h.\text{base}} 42) * (\neg h'.\text{valid} \vee \xmapsto{\text{wss}}_{h'.\text{base}} 0)\}$

$\{\textbf{trap} \vee g \xmapsto{\text{wg}} - * \xmapsto{\text{wss}}_{h.\text{base}} 42 * h' \in \mathcal{V}[\![\textbf{handle}]\!]\}$

`Call an adversary function` $[\textbf{handle}] \rightarrow []$ `with arg` h'

Robust example

$\{g \xmapsto{\text{wg}} -\}$

`Allocate two handles` h `and` h'

$\{g \xmapsto{\text{wg}} - * (\neg h.\mathsf{valid} \vee \xmapsto{\text{wss}}_{h.\mathsf{base}} 0) * (\neg h'.\mathsf{valid} \vee \xmapsto{\text{wss}}_{h'.\mathsf{base}} 0)\}$

`Write 42 to` h

$\{g \xmapsto{\text{wg}} - * (\mathbf{trap} \vee \xmapsto{\text{wss}}_{h.\mathsf{base}} 42) * (\neg h'.\mathsf{valid} \vee \xmapsto{\text{wss}}_{h'.\mathsf{base}} 0)\}$

$\{\mathbf{trap} \vee g \xmapsto{\text{wg}} - * \xmapsto{\text{wss}}_{h.\mathsf{base}} 42 * h' \in \mathcal{V}[\![\mathbf{handle}]\!]\}$

`Call an adversary function` $[\mathbf{handle}] \to []$ `with arg` h'

$\{\mathbf{trap} \vee g \xmapsto{\text{wg}} - * \xmapsto{\text{wss}}_{h.\mathsf{base}} 42 * h' \in \mathcal{V}[\![\mathbf{handle}]\!]\}$

Robust example

$\{g \xmapsto{\text{wg}} -\}$

`Allocate two handles` h `and` h'

$\{g \xmapsto{\text{wg}} - * (\neg h.\text{valid} \vee \xmapsto{\text{wss}}_{h.\text{base}} 0) * (\neg h'.\text{valid} \vee \xmapsto{\text{wss}}_{h'.\text{base}} 0)\}$

`Write 42 to` h

$\{g \xmapsto{\text{wg}} - * (\textbf{trap} \vee \xmapsto{\text{wss}}_{h.\text{base}} 42) * (\neg h'.\text{valid} \vee \xmapsto{\text{wss}}_{h'.\text{base}} 0)\}$

$\{\textbf{trap} \vee g \xmapsto{\text{wg}} - * \xmapsto{\text{wss}}_{h.\text{base}} 42 * h' \in \mathcal{V}[\![\textbf{handle}]\!]\}$

`Call an adversary function` $[\textbf{handle}] \to []$ `with arg` h'

$\{\textbf{trap} \vee g \xmapsto{\text{wg}} - * \xmapsto{\text{wss}}_{h.\text{base}} 42 * h' \in \mathcal{V}[\![\textbf{handle}]\!]\}$

`Read from` h

`Set global variable` g `to the read value`

`Free` h

$\{\textbf{trap} \vee g \xmapsto{\text{wg}} 42 * h' \in \mathcal{V}[\![\textbf{handle}]\!]\}$

Robust example

```
$\{g \xmapsto{\text{wg}} -\}$
Allocate two handles $h$ and $h'$
$\{g \xmapsto{\text{wg}} - * (\neg h.\text{valid} \vee \xmapsto{\text{wss}}_{h.\text{base}} 0) * (\neg h'.\text{valid} \vee \xmapsto{\text{wss}}_{h'.\text{base}} 0)\}$
Write 42 to $h$
$\{g \xmapsto{\text{wg}} - * (\textbf{trap} \vee \xmapsto{\text{wss}}_{h.\text{base}} 42) * (\neg h'.\text{valid} \vee \xmapsto{\text{wss}}_{h'.\text{base}} 0)\}$
$\{\textbf{trap} \vee g \xmapsto{\text{wg}} - * \xmapsto{\text{wss}}_{h.\text{base}} 42 * h' \in \mathcal{V}[\![\textbf{handle}]\!]\}$
Call an adversary function $[\textbf{handle}] \to []$ with arg $h'$
$\{\textbf{trap} \vee g \xmapsto{\text{wg}} - * \xmapsto{\text{wss}}_{h.\text{base}} 42 * h' \in \mathcal{V}[\![\textbf{handle}]\!]\}$
Read from $h$
Set global variable $g$ to the read value
Free $h$
$\{\textbf{trap} \vee g \xmapsto{\text{wg}} 42 * h' \in \mathcal{V}[\![\textbf{handle}]\!]\}$
$\{\textbf{trap} \vee g \xmapsto{\text{wg}} 42\}$
```

Scaling up: stack module

Stack module
Client module

- S defines library functions
- C imports functions from S
- Use segment memory to store stacks
- Showcase Iris-MSWasm on a larger example
- Contrast with linear memory stack module

Stack module: Iris-Wasm vs Iris-MSWasm

$Mymodule:

```
Create a stack s
Push 42 and 10 onto s
Map $advf onto s
Set x := length(s)
Assert x = 2
```

In Iris-Wasm:

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

Stack module: Iris-Wasm vs Iris-MSWasm

$Mymodule:

```
Create a stack s
Push 42 and 10 onto s
Map $advf onto s
Set x := length(s)
Assert x = 2
```

In Iris-MSWasm:

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

Conclusion

WebAssembly is perfect for Iris

- Formally defined
- Industrial scale

Iris is perfect for WebAssembly

- Showcase strengths
- Iron out extension proposals

Should all extensions of WebAssembly come with an attached mechanised program logic?