SecWasm: Information Flow Control for WebAssembly

Iulia Bastys[1], Maximilian Algehed[1], Alexander Sjösten[2], and Andrei Sabelfeld[1]

[1] Chalmers University of Technology
[2] TU Wien

Abstract. We introduce SecWasm, the first general purpose information-flow control system for WebAssembly (Wasm), thus extending the safety guarantees offered by Wasm with guarantees that applications manipulate sensitive data in a secure way. SecWasm is a hybrid system enforcing termination-insensitive noninterference which overcomes the challenges posed by the uncommon characteristics for machine languages of Wasm in an elegant and thorough way.

1 Introduction

WebAssembly (Wasm) [22] is gaining popularity as a new standard for near-native low-level code and is becoming a popular compilation target for languages like C, C++, and Rust. Designed to enable high-performance web applications, Wasm is currently supported by all major browsers [48]. Wasm also boasts support to standalone environments such as Node.js and it has been deployed for decentralized cloud computing [24], smart contracts [1], and IoT [51,40].

Consider a password meter website *PM* which needs to communicate with a third-party website *TP* to fetch a password dictionary. *PM* would fetch the dictionary in the beginning and signal the end of a successful run at the end. Current Wasm security guarantees are able to prevent direct exfiltration, but cannot ensure the password is not leaked (through URL parameter encoding or otherwise) given a malicious developer providing module *PM*.

More specifically, Wasm security relies on the browser's same-origin policy and a memory-safe sandboxed execution environment [2] with separate memory and code space [22]. Wasm has an unstructured linear memory which can be grown dynamically. To ensure memory safety, all memory accesses are dynamically checked against the memory bounds, trapping any out-of-bounds access. Furthermore, Wasm applications have structured control flow, therefore disallowing jumps to arbitrary locations. In this way, Wasm ensures *control-flow integrity* (CFI) [3], such that Wasm code can be compiled and validated in a single pass.

While Wasm offers CFI, it remains an open challenge to ensure a *secure flow of information* through its applications. A promising technique addressing this is *information-flow control* (IFC) [36], which tracks both explicit and implicit information flows. While first valuable steps have been taken in this direction [49,43,19,41], prior work is yet to address implicit flows [19,41], provide formal guarantees [43,19], handle flows via the memory [41], or apply beyond specialized scenarios of constant-time Wasm for cryptographic algorithms [49].

A *general* and *sound IFC* approach to Wasm suitable for *general-purpose applications* is pending. Moreover, it is a prerequisite for further progress in IFC techniques for WebAssembly. Although several IFC systems for other machine languages have been proposed [10,25,20,11,7,5,30,29,13,52,21], they cannot be immediately repurposed here. Wasm is not a regular low-level language. Its *structured* control flow mechanisms and *unstructured* linear memory are uncommon. And when it comes to IFC, they prove to be quite challenging on certain aspects.

The structured control flow allows us to design an IFC system which leverages Wasm's syntax to compute the control flow regions directly. This in contrast to IFC approaches for other machine languages which resorted to employing external tools [10,5,52,25,13] or adding artificial syntactic constructs [29,13,52] to achieve some structure at the low-level. However, Wasm's handling of the operand stack which, to the best of our knowledge, is unique among machine languages requires some innovation when it comes to defining the security properties enforced by the IFC system.

Dealing with an unstructured linear memory entails an analysis in itself, not only on what labeling tactic to apply, but also on what type of IFC enforcement to design—both quite intermingled. While choosing the type of enforcement may seem trivial, choosing the right memory labeling approach does not. When it comes to the former, the reasoning is straightforward. On the one hand, Wasm's well-developed type system makes it suitable for static IFC. On the other hand, managing dynamic flows such as memory accesses statically would lead to a restrictive and rigid system, tipping the balance in favor of dynamic IFC. Yet, a purely dynamic IFC approach usually bearing significant execution overhead is not necessary for Wasm, since the language does not exhibit dynamic features. Thus, the challenge remains in labeling the memory such that it minimizes the dynamic checks while still maintaining permissiveness and expressiveness.

In this paper, we propose SecWasm, a hybrid IFC system addressing the challenges above in an elegant and thorough way. As is common [13,52,29,25,5,49,10], our focus is on *confidentiality*, with the security goal of preventing information from secret inputs to leak to public outputs. However, we envision our mechanisms to be suitable for tracking some facets of integrity, thanks to the duality of confidentiality and information-flow integrity [12].

Non-goals To delimit the scope of the paper, we emphasize the non-goals of SecWasm, pertaining to handling the sources of non-determinism in WebAssembly: lack of bit pattern for NaN values, resource exhaustion, and imported host functions [22]. While we acknowledge that non-determinism can lead to illicit information-flows through side channels (e.g., via the micro-architectural state of the processor [44], or termination and progress channels [4]), we consider it a worthwhile subject for future work and not crucial for laying the foundations of general IFC in Wasm, which is the goal of this paper.

Contributions In brief, we make the following contributions:

– We discuss the key aspects of IFC for Wasm, to back up and give an intuition for the design of SecWasm (Section 3).
– We present SecWasm, the first general IFC system for Wasm (Section 4).
– We formally prove SecWasm to enforce termination insensitive noninterference (Section 5).

2 Background on Wasm

This section gives a brief overview of the Wasm specifics required to understand SecWasm. In particular, we present the basic features and discuss important aspects such as *structured control flow*, *linear memory*, and *security characteristics*. For more details on Wasm, we refer the reader to the initial publication [22] or official live documentation [50]. In the following and the rest of the paper, we focus on Wasm v1.0 [47].

2.1 Basics

We begin by presenting the syntactic features of WebAssembly most relevant for SecWasm (Figure 1).

Modules Wasm programs are organized into modules. A module is composed of a list of function types, a list of functions, a table identifying function pointers with functions, a linear memory of raw bytes[3], and a list of typed global variables.

A module is instantiated through an embedder, which is a host environment usually attached to the JavaScript engine in a web browser. When instantiating a module, the embedder must provide definitions for everything that should be imported, such as host functions, and an initial linear memory m. The module can also export Wasm functions the embedder can invoke, and the embedder can read the linear memory of the module.

Each function *func* has a type specifying its signature by reference to a function type defined in the module. Functions may have local variables and consist of a sequence of instructions comprising the function body. Functions are not first-class, meaning they cannot be used as arguments to or returned from other functions, nor assigned to variables. However, functions can call other functions, including themselves recursively. Functions can be invoked directly using the **call** instruction which takes as argument the index of the function in the functions vector, or indirectly with the **call_indirect** instruction via the function pointer table *tbl* mapping integers to functions.

Global variables *gbl* may be either mutable or immutable and are in scope to the entire module. Local variables are always mutable and only in scope to the executing function.

Types Wasm supports four primitive value types t: 32 and 64-bit integers (i32 and i64) and single and double precision floating-point numbers (f32 and f64). Complex data types such as arrays or pointers do not exist in Wasm, and any

[3] Wasm 1.0 only has support for a single memory per module.

$$
\begin{array}{lll}
\text{(modules)} & \mathit{module} & ::= \{\mathsf{types}\ \mathit{ft}^*, \mathsf{funcs}\ \mathit{func}^*, \mathsf{tables}\ \mathit{tbl}, \mathsf{mems}\ m^1, \mathsf{globals}\ \mathit{glb}\} \\
\text{(functions)} & \mathit{func} & ::= \{\mathsf{type}\ \mathit{idx}, \mathsf{locals}\ t^*, \mathsf{body}\ \mathit{expr}\} \\
\text{(immediates)} & i & ::= \mathit{nat} \\
\text{(value types)} & t & ::= \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{f32} \mid \mathsf{f64} \\
\text{(global types)} & \mathit{gt} & ::= \mathsf{mut}^?\ t \\
\text{(function types)} & \mathit{ft} & ::= t^* \to t^* \\
\text{(block types)} & \mathit{bt} & ::= t^* \to t^* \\
\text{(constants)} & k & ::= \ldots \\
\text{(instructions)} & \mathit{instr} & ::= \mathit{data} \mid \mathit{mem} \mid \mathit{ctrl} \mid \mathit{admin} \\
 & \mathit{data} & ::= t.\mathbf{\mathsf{const}}\ n \mid t.\mathit{unop} \mid t.\mathit{binop} \mid \mathbf{\mathsf{drop}} \mid \mathbf{\mathsf{select}} \mid \mathbf{\mathsf{local.get}}\ i \mid \mathbf{\mathsf{local.set}}\ i \\
 & & \quad \mid\ \mathbf{\mathsf{local.tee}}\ i \mid \mathbf{\mathsf{global.get}}\ i \mid \mathbf{\mathsf{global.set}}\ i \\
 & \mathit{mem} & ::= t.\mathbf{\mathsf{load}}\ a\ o \mid t.\mathbf{\mathsf{store}}\ a\ o \mid \mathbf{\mathsf{memory.size}} \mid \mathbf{\mathsf{memory.grow}} \\
 & \mathit{ctrl} & ::= \mathbf{\mathsf{nop}} \mid \mathbf{\mathsf{unreachable}} \mid \mathbf{\mathsf{block}}\ (\mathit{bt})\ \mathit{expr}\ \mathbf{\mathsf{end}} \mid \mathbf{\mathsf{loop}}\ (\mathit{bt})\ \mathit{expr}\ \mathbf{\mathsf{end}} \\
 & & \quad \mid\ \mathbf{\mathsf{if}}\ (\mathit{bt})\ \mathit{expr}\ \mathbf{\mathsf{else}}\ \mathit{expr}\ \mathbf{\mathsf{end}} \mid \mathbf{\mathsf{br}}\ i \mid \mathbf{\mathsf{br\_if}}\ i \mid \mathbf{\mathsf{br\_table}}\ i^+ \mid \mathbf{\mathsf{return}} \mid \mathbf{\mathsf{call}}\ i \\
 & & \quad \mid\ \mathbf{\mathsf{call\_indirect}}\ \mathit{ft} \\
 & \mathit{admin} & ::= \mathbf{\mathsf{trap}} \mid \mathbf{\mathsf{label}}_n\{\mathit{expr}\}\ \mathit{expr}\ \mathbf{\mathsf{end}} \mid \mathbf{\mathsf{frame}}_n\{\mathit{frame}\}\ \mathit{expr}\ \mathbf{\mathsf{end}} \mid \mathbf{\mathsf{invoke}}\ a \\
\text{(expressions)} & \mathit{expr} & ::= \mathit{instr} \mid \mathit{expr}; \mathit{expr}
\end{array}
$$

Fig. 1: Selected Wasm abstract syntax. Non-empty sequences are denoted with exponent $^+$, possibly empty ones with exponent *, possibly empty singleton sequences with exponent ^{1}, and optional arguments with exponent $^?$.

representation of these types in the source language is compiled down to a primitive type. Function types *ft* (as well as block types *bt*) define a sequence of Wasm values taken as parameters and a sequence of values to return.

Instructions Wasm bytecode is executed as a stack-machine, where instructions pop argument values off and push result values onto an operand stack.

Instructions are partitioned into *data*, *mem*, *ctrl*, and *admin*. Data instructions either manipulate the operand stack directly (t.**const** n, **drop**, **select**), the local variables (**local.get** i, **local.set** i, **local.tee** i), or the global variables (**global.get** i, **global.set** i). Memory instructions are used for interaction with the linear memory. Instructions **store** and **load** write to and read from the linear memory, respectively. **memory.size** returns the current size of the memory, and **memory.grow** extends it dynamically. Control instructions comprise scoping constructs (**block**), loops (**loop**), conditionals (**if**), structured unconditional (**br**, **br_table**, **return**) and conditional jumps (**br_if**), and direct (**call**) and indirect function calls (**call_indirect**). Finally, **nop** does nothing, while **unreachable** causes an unconditional, uncatchable trap exception. When a trap occurs, the entire computation is aborted, and no other changes to the state are allowed. Wasm does not handle the traps, but propagates them to the embedder. Traps are expressed by the administrative instruction **trap**. Other *admin* instructions express reduction of control instructions. As such, **block**, **loop**, and **if** reduce to **label**s, and **call**s to **invoke**, which further reduce to **frame**s. Labels $\mathbf{label}_n\{\mathit{expr}_1\}\ \mathit{expr}_2$ **end** carry the return arity n of the block, the block's body expr_2, and the continuation expr_1 to execute when a jump occurs within the block. **invoke** represents the invocation of a function instance identified by its address a. Finally, frames $\mathbf{frame}_n\{\mathit{frame}\}\ \mathit{expr}$ **end** carry the return arity n and body *expr* of the function and the values of its arguments stored in *frame*.

2.2 Structured Control Flow

Unlike other machine languages, the control flow in Wasm is structured and this guarantees a program cannot jump to arbitrary locations. The structured control flow is obtained by a combination of nested block constructs and jumping instructions permitted only from within the blocks, and only as far out as the nesting depth allows.

Blocks Blocks are formed by standard control flow constructs **if** and **loop**, and scoping construct **block**. Each such construct terminates with an **end** opcode indicating where the construct's lexical scope ends.

Branches Wasm further implements its structured control flow with several branching instructions: **br**, **br_table**, and **return**—unconditional, and **br_if**—conditional. The crux of these branching instructions is that unlike unstructured control flow, such as `goto` in C, they can only be executed inside nested blocks. Branches have *label* immediates referencing outer blocks by their relative nesting depth. This makes the labels scoped and able to reference only constructs in which their corresponding branches are nested. Depending on the type of construct, the effect of taking a branch differs. For a **block** or **if** instruction, a *forward* jump occurs that resumes execution *after* the matching **end**. On the other hand, a **loop** has a *backward* jump that *restarts* the loop.

Operand Stack Unwinding In Wasm, the operand stack contains three types of entries: values $t.\textsf{const}\ n$, labels $\textsf{label}_n\{expr\}$, and frames $\textsf{frame}_n\{frame\}$, with the latter two modeled by their respective administrative instructions. As such, when a block (or **call**) instruction executes, the top values corresponding to the block (or function) arguments are temporarily popped, a label (or frame) is pushed, and the value arguments are pushed back, order preserved.

Branching retains the values on top of the operand stack corresponding to the return values of the current block (but also to the argument values of the continuation) and pops *all* entries off the stack until and including the label entry corresponding to the continuation. Basically, this amounts to popping a number of labels off the stack equal to branching immediate +1 and all other value entries in between.

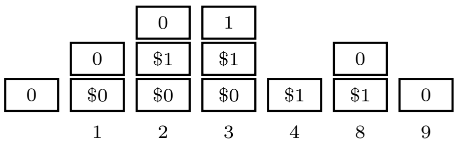
A **return** from a function keeps the top values on the stack denoting the function return values and pops everything off the stack until and including the first frame, which represents the frame of the current function.

Example Consider the code in Figure 2a and assume an initial operand stack containing only value i32.**const** 0. The evolution of the stack during the execution of the code is depicted in Figure 2b. In the following, we will go through each instruction in the code of Figure 2a and explain the behavior of the stack. Blocks are labeled \$0 and \$1 for easier referencing.

Note the type of block \$0 is $\textsf{i32} \rightarrow \textsf{i32}$. This means the block takes one argument and has only one return value, both of type i32. More specifically, before entering and leaving the block, the operand stack requires on top a value

```
1 block (i32 → i32) $0
2    block (i32 → ε) $1
3       i32.eqz
4       br_if 0
5       i32.const 1
6       br 1
7    end
8    i32.const 0
9 end
```

(a)



(b)

Fig. 2: Branching example (a) and the evolution of the operand stack during its execution (b). The stack and index i below denote the operand stack after the execution of the instruction on line i. Values are depicted as n instead of t.**const** n. \$0 = **label**$_1\{\varepsilon\}$; \$1 = **label**$_0$\{i32.**const** 0\};

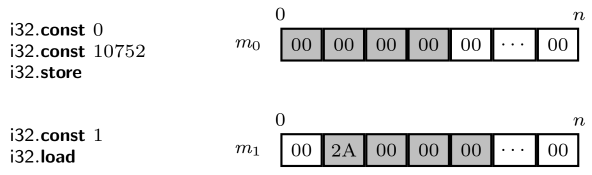
of type i32. Block \$1 of type i32 $\to \varepsilon$ only takes an argument of type i32 and has no return values.

When block \$0 is entered, value i32.**const** 0 is popped off the stack, label **label**$_1$\{i32.**const** 0\} is pushed, then i32.**const** 0 is pushed back in. The same behavior arises for instruction 2. i32.**eqz** pops the top value off the stack and checks if it equals 0. It does, so it pushes back i32.**const** 1, otherwise it would have pushed i32.**const** 0. **br_if** 0 is a conditional jump which executes if the top of the operand stack is i32.**const** 1. It is (step 3), so control is given to the instruction at the end of block \$1. When this happens, the label of block \$1 is popped off the stack. Note i32.**const** 1 was popped off during the execution of **br_if** 0. Instruction 8 simply pushes i32.**const** 0 on the operand stack. Since block \$0 needs to return an i32 value, when leaving it on line 9, i32.**const** 0 is temporarily popped off, the block label is removed and i32.**const** 0 is pushed back in.

2.3 Linear Memory

The main storage for a Wasm program is an unmanaged linear memory representing a contiguous mutable array of raw bytes [50] which uses the little-endian byte order [22]. The memory is instantiated with an initial size and initialized with zeros. It can be grown dynamically with instruction **memory.grow** and queried for the current size with **memory.size**. The memory can be accessed through **load** and **store** instructions, with the addresses being unsigned integers of type i32. Whenever a memory access occurs, a dynamic check ensures the address is within the memory bounds. If it is not, a trap occurs.

Writing to and Reading from Memory Figure 3 depicts instances of memory access. Initially, linear memory m_0 of size **memory.size** $= n$ contains only zeros. We store 32-bit integer 10752 on array positions 0 to 3, as the value takes four bytes, and get a new memory m_1. Reading a 32-bit integer from m_1 (starting) at location 1 means converting bytes 2A000000 to 42. Observe bytes from values stored at adjacent positions in the memory can be interpreted as a new value, as the raw data in the memory can be used to represent other numbers [50].



```
i32.const 0
i32.const 10752
i32.store

i32.const 1
i32.load
```

Fig. 3: Illustrative memory accesses for reads and writes. Highlighted memory locations denote the positions in the memory array where the value is written to/read from.

Security Specifications The linear memory is disjoint from the code space, the execution stack, and the runtime engine's data structures. As the memory is unmanaged, Wasm does not provide garbage collection. Moreover, being the only unmanaged part of Wasm, the linear memory becomes the only component of the execution environment prone to corruption by buggy or malicious Wasm code. Thus, untrusted Wasm code can safely execute in the same address space as other code.

Unfortunately, this does not do away with buggy programs susceptible to attacks *via* the memory. Specifically, certain memory vulnerabilities in C code can persist when compiled to Wasm [27]. While these vulnerabilities do not allow the attacker to corrupt the execution environment, meaning they are *memory-safe*, they can still lead to insecure information flows that, e.g., may breach confidentiality; in other words, they are *information-flow unsafe*.

3 Challenges and Design Choices

Next, we highlight the challenges arising from building an IFC system for Wasm and give an intuition for the design choices taken when modeling it.

3.1 Attacker Model

As usual when designing an IFC system, we consider a join semi-lattice $(\mathcal{L}, \sqsubseteq)$ of security levels ℓ, where data at level $\ell_d \in \mathcal{L}$ can flow to an observer at level $\ell_o \in \mathcal{L}$ if and only if $\ell_d \sqsubseteq \ell_o$.

The attacker is thus able to observe information below their security level $\mathcal{A}$. In addition, they have the ability to execute a Wasm program, and have access to the final state of the global variables whose labels ℓ may flow to $\mathcal{A}$ ($\ell \sqsubseteq \mathcal{A}$). The attacker does not have access to the linear memory, nor to the operand stack after the execution of the Wasm program. However, as customary, in our noninterference proofs we also show $\mathcal{A}$-equivalence on the operand stacks and linear memories of two runs to get the appropriate induction invariants.

While these requirements may seem restrictive, they are in line with previous work [10] and we argue our model allows for a realistic attacker, external to the system in which the Wasm code is running. Recall the attacker providing the malicious *PM* module in the password meter example in the introduction. The attacker is able to supply malicious Wasm code, but cannot control the surrounding JavaScript context, is able to see external events (such as web requests) emanating from the Wasm code, but cannot usurp the entire surrounding execution context and thus cannot see the whole linear memory at the end of the execution. As Wasm does not have a notion of web requests or channel communication with the surrounding execution context, we model external events by the final value of global variables.

Finally, as already mentioned, we ignore information leaks stemming from other side channels or from the interaction with the environment.

3.2 Unstructured Linear Memory

When it comes to the linear memory, we point out three properties we want our IFC enforcement to fulfill, all necessary to achieve a more expressive and permissive system. The system should: 1) handle dynamic data structures compiled down from the high-level language, such as objects and arrays; 2) allow for a dynamic memory reuse; and 3) provide an IFC-sound memory.

In addition, for the IFC enforcement *per se*, two aspects need to be considered: type of enforcement and memory labeling strategy (including granularity and sensitivity). While tightly bound, we address them separately in the following paragraphs.

Type of IFC Enforcement In theory, we could model our system as a static, dynamic, or hybrid enforcement. In practice, enforcing IFC in Wasm dynamically could be an overkill since the language does not have dynamic features, e.g., in the style of JavaScript[4]. Leveraging Wasm's type system and building a fully static IFC enfocement is not an option either because of the unstructured nature of the memory. Statically, we do not have access to the memory address we are reading from/writing to, so we cannot propagate memory taints via the type system. A static enforcement can be indeed forced by either labeling the entire memory upfront, or by using one memory for every security level in the lattice, as previously suggested [49]. However, the former approach leads to a rigid system breaking points 1) and 2), while the latter suffers from several drawbacks. Firstly, it does not scale well to larger lattices and secondly, objects in the high-level language with differently labeled fields would have to be split across different memories. Finally, handling implicit flows in a meaningful way is not obvious.

Thus, the solution we adopt in this paper is hybrid IFC enforcement. More specifically, we design a mainly static enforcement augmented with dynamic security checks on memory access instructions. This is consistent with previous work on IFC for other low-level languages without dynamic features [25,5,30,29,13,52], which are fully static as they do not handle a linear memory, but rely entirely on a heap. Hybrid IFC systems have also been discussed for TAL-like languages [21]

[4] Wasm does exhibit some dynamism through `importObject`, but since we do not handle imported host functions in this paper, we do not consider it further here.

Example 1.

```
i32.const 1
i32.load L
```

Example 2.

```
i32.const 1
i32.load H
```

Example 3.

```
i32.const 2
i32.const c
i32.store H
```

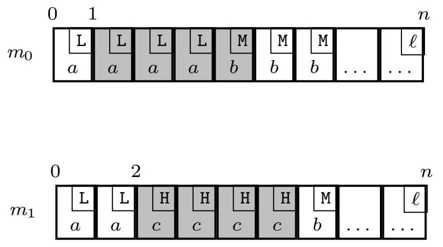


Fig. 4: Illustrative examples for memory access rules. Locations $\boxed{\overset{\texttt{L}}{a}}$ denote bytes of value a labeled L. Highlighted locations are read from/written to.

and even JavaScript [38,23], the former to increase expressiveness of previous static enforcements, the latter to reduce the overhead of the dynamic monitor.

Labeling the Linear Memory Recall Wasm's linear memory is a contiguous array of raw bytes. To achieve more flexibility, we opt for a fine-grained approach of labeling the memory and assign a label to every memory location. As such, each memory location l maps in SecWasm to a pair (b, ℓ) of byte b and security level ℓ.

The fine-grained labeling allows for a straightforward handling of arrays and objects when compiled down to Wasm, as they can occupy a contiguous sequence of memory locations, instead of non-adjacent ranges of locations (a first step in satisfying point 1). For the same reason, but also for satisfying point 2), we pursue a flow-sensitive approach. Flow-insensitivity would again require the memory to be statically labeled upfront, without possibility of changing its taints. As mentioned earlier, this is a rigid approach we do not consider further.

Security Considerations One consequence of these choices is that memory access instructions become adorned with a security label ℓ. Then t.**load** ℓ (t.**store** ℓ) reads from (writes to) the memory a value of type t and security level ℓ.

Further, to reduce the dynamic overhead, we employ dynamic checks only when reading from the memory. Checks when writing to the memory are not needed. First, because the labels in the memory are updated upon a write, and second, because the security type system ensures the security labels of the value to be written, of the execution context, and of the address to write at all have lower sensitivity than the instruction's label. As such, while writing to memory will always succeed, given the instruction does not trap due to insufficient resources, reading from memory needs to additionally ensure the security labels of all memory locations required to form the value read are below level ℓ of the instruction. Thus, given memory m_0 in Figure 4, the program in Example 1 will trap ($\texttt{M} \not\sqsubseteq \texttt{L}$), while the one in Example 2 will not ($\texttt{L} \sqcup \texttt{M} \sqsubseteq \texttt{H}$). Finally, executing the program in Example 3 with memory m_0 produces memory m_1.

Another consequence of our memory labeling strategy is that *new* memory locations require a security label as well. (Recall Wasm's memory can be extended dynamically with construct **memory.grow**.) Thus, for security reasons the newly created memory locations are labeled with the bottom label `L` of the lattice.

Moreover, calls to **memory.grow** can only take place in public contexts and by a public value. Allowing other levels would leak private information, as depicted in the code snippets in Example 4 and Example 5. In both examples, by comparing the global values stored at positions 0 and 1 in the final state, the attacker can learn the secret read on line 3 in Example 4, respectively line 4 in Example 5.

Example 4.

```
1 memory.size
2 global.set 0
3 i32.load H
4 memory.grow
5 memory.size
6 global.set 1
```

Example 5.

```
1 memory.size
2 global.set 0
3 i32.const 1
4 i32.load H
5 if (memory.grow)
6 else (i32.const 0)
7 memory.size
8 global.set 1
```

3.3 Structured Control Flow

One of the challenges of extending Wasm with IFC is computing the control flow regions for handling implicit flows.

Wasm has scoped control flow instructions, similarly to high-level languages, and branching instructions which extend their lexical scope, similarly to other low-level languages. Computing the scope extension is what sets SecWasm apart, as employing external tools or performing additional computations [5,10] does not seem to be necessary for it. Instead, we benefit from branching instructions arising only within *nested* blocks and use their immediates to compute the scope extension.

Consider the code snippet in Example 6. It contains three nested **block**s (labeled \$B0-\$B2 and whose types we omit for clarity) and two conditional branching instructions inside block \$B2, with **br_if** 1 (line 8) extending \$B2's scope until the end of block \$B1. The first branch (line 8) is conditioned by the medium-labeled value read on line 7. Then, instructions on lines 8-13 will be in medium context. However, since the second branch (line 10) is conditioned by the high-labeled value read on line 9, the execution of instructions on lines 10-11 will be in high context. We assume $expr_n$, with $0 \leq n \leq 4$, are not branching instruction. Note $expr_4$ is not highlighted in red, nor $expr_5$ in blue. The reason for this is that $expr_4$ is executed irrespective of whether $expr_3$ gets executed or not. Similarly, $expr_5$ is not in a medium context as it is always executed.

Example 6.

```
1  block $B0
2    expr0
3    block $B1
4      expr1
5      block $B2
6        expr2
7        t.load M
8        br_if 1
9        t.load H
10       br_if 0
11       expr3
12     end
13     expr4
14   end
15   expr5
16 end
17 expr6
```

In brief, immediate i of a branching instruction extends the scope of the current block until the end of the ith-1 block, where counting starts at 0 from the current block. We further use this information to compute the control flow regions without resorting to other additional tools.

The pc upgrading and downgrading around the control flow regions is not surprising, and this is usually dealt with by adopting a stack of security levels [53], with the top pc being the effective one. We follow a similar tactic and push a pc entry onto the stack whenever we enter a block. What SecWasm does differently next, is to use a *flow-sensitive* stack, i.e., a stack whose entry sensitivity can change

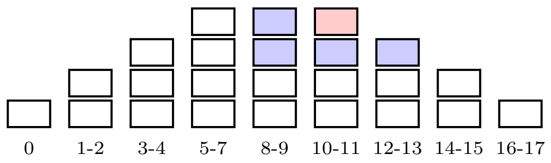


Fig. 5: *pc* stack progression for Example 6. Indices denote code line numbers, white denotes a low program counter, blue medium, and red high.

during typing (Figure 5), in contrast to most previous approaches employing a flow-insensitive one. More specific details on this are discussed in Section 4.3.

3.4 $\mathcal{A}$-Equivalences

The final challenge we face is not to ensure the design of SecWasm is sound, information flow in Wasm is comparatively straight forward, but *proving* it is sound. A first step in this direction is coming up with the *right* definitions to get the appropriate induction invariants for proving noninterference.

While we are interested in global variables equivalence with respect to the attacker (Section 3.1), we need to show some kind of $\mathcal{A}$-equivalence holds throughout the program's execution for other parameters as well, such as memory and operand stack, even though the attacker *does not have access to them.*

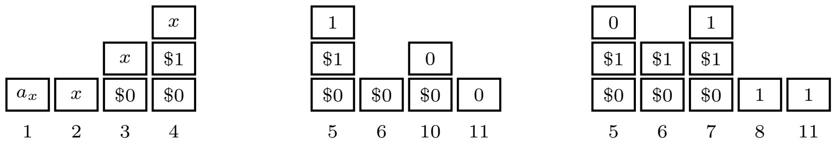
Memory $\mathcal{A}$-Equivalence Traditionally, ℓ-equivalence on memories m_0 and m_1 (denoted $m_0 \sim_\ell m_1$) is defined such that for every memory location l, if $m_0(l) = (k_0, \ell_0)$ and $m_1(l) = (k_1, \ell_1)$ and both $\ell_0, \ell_1 \sqsubseteq \ell$, then $k_0 = k_1$ and $\ell_0 = \ell_1$.

However, this relation is not an equivalence relation, as it is not transitive. Given memories $m_1 = \{0 \mapsto (1, \mathtt{L}), 1 \mapsto (1, \mathtt{L}), 2 \mapsto (3, \mathtt{H})\}$, $m_2 = \{0 \mapsto (1, \mathtt{L}), 1 \mapsto (1, \mathtt{H}), 2 \mapsto (2, \mathtt{H})\}$, and $m_3 = \{0 \mapsto (1, \mathtt{L}), 1 \mapsto (2, \mathtt{L}), 2 \mapsto (1, \mathtt{H})\}$, $m_1 \sim_{\mathtt{L}} m_2$ and $m_2 \sim_{\mathtt{L}} m_3$, but $m_1 \not\sim_{\mathtt{L}} m_3$. Due to this, the classical formulation for confinement will not be strong enough to hold true, as after typing a program in a high context, executing it will not necessarily result in ℓ-equivalent memories. Because of the flow-sensitivity, the program execution in a high context is confined to strictly making more memory locations secret.

This means we need a stronger relation for memories, an ordered-equivalence $\blacktriangleleft_{\mathcal{A}}$ which says two memories m_0 and m_1 are $\blacktriangleleft_{\mathcal{A}}$-equivalent if m_1 has strictly more high-labeled indices and all low-labeled indices are the same between m_0 and m_1 (see Definition 6 in Section 5).

Operand Stack $\mathcal{A}$-Equivalence Defining $\mathcal{A}$-equivalence for two unwinding operand stacks is more involved.

Consider the Wasm code in Example 7 prepending the code in Figure 2a with instructions 1-2 for reading value of secret $x_{\mathtt{H}}$. This also corresponds to C code `if (`$x_{\mathtt{H}}$`) {return 0;} else {return 1;}`. Figure 6 depicts the evolution of the operand stack during the execution of this program for both cases when $x_{\mathtt{H}} = 0$ and $x_{\mathtt{H}} \neq 0$.



(a) $x_{\mathtt{H}} = 0$ (b) $x_{\mathtt{H}} \neq 0$

Fig. 6: Evolution of the operand stack for Example 7. The stack and index i below denote the operand stack after the execution of the instruction on line i. Values are depicted as n instead of $t.$**const** n. $\$0 = \textbf{label}_1\{\epsilon\}$; $\$1 = \textbf{label}_0\{\text{i32}.\textbf{const } 0\}$; x is the value read from memory starting at location a_x.

Example 7.

```
1  i32.const a_x
2  i32.load H
3  block (i32 → i32) $0
4     block (i32 → ε) $1
5        i32.eqz
6        br_if 0
7        i32.const 1
8        br 1
9     end
10    i32.const 0
11 end
```

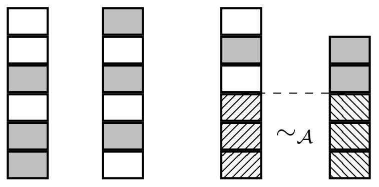
Since we consider x to be high, running the program with values for $x_{\mathtt{H}}$ from the two cases gives us two different operand stacks which at the end of the execution must be indistinguishable to an attacker. We say the end of the execution since instructions 6-11 will be in high context. (**br_if** 0 sets a high context for instructions 6-9 and **br** 1 on line 8 extends it until line 11.)

Generally, we show this indistinguishability by first relating through an equivalence relation $\sim_{\mathcal{A}}$ two operand stacks with the same shape OS_1 and OS_2 and second, by relating through an ordered equivalence $\blacktriangleleft_{\mathcal{A}}$ and a *confinement* lemma two operand stacks OS_1 and OS'_1 (OS_2 and OS'_2, respectively) when entering and leaving a high-context area. Finally, a *triangle* lemma proves the two final operand stacks OS'_1 and OS'_2 $\mathcal{A}$-equivalent.

$$\begin{array}{ccc} OS_1 & \blacktriangleleft_{\mathcal{A}} & OS'_1 \\ \sim_{\mathcal{A}} & & \sim_{\mathcal{A}} \\ OS_2 & \blacktriangleleft_{\mathcal{A}} & OS'_2 \end{array}$$

Recall the elements on the operand stack are values, frames, and labels, and none of which contains security levels. Before relating the operand stacks in attacker-equivalence relations, we need to relate them to another structure containing security levels, and this is a type stack TS of labeled types $t\langle\ell\rangle$. Then, $TS \Vdash OS$ (Definition 3 in Section 5) says that OS is in agreement with TS, meaning that if disconsidering frames and labels, then for every labeled type $t\langle\ell\rangle$ in TS there is a corresponding value $t.$**const** k on the same position in OS.

Defining relation $\sim_{\mathcal{A}}$ simply means ensuring the operand stacks satisfy certain requirements given their corresponding labeled type stacks. Figure 7a illustrates this relation. Cells denote values on the operand stack, and gray cells denote values whose corresponding labeled type on the type stack has a high label. Basically, $\sim_{\mathcal{A}}$ says that any two operand stacks of the same shape (without frames and labels) and with equal low values (the label of the corre-



(a) $OS \sim_{\mathcal{A}} OS'$ (b) $OS \blacktriangleleft_{\mathcal{A}} OS'$

Fig. 7: Operand stack equivalence relations in SecWasm. White is low, gray is high, striped is either.

sponding type is low) on the same positions are attacker-equivalent (Definition 4 in Section 5).

Defining relation $\blacktriangleleft_{\mathcal{A}}$ is particularly challenging, as we need to specify what happens to the operand stack during the high-context execution. If it unwinds, how much does it unwind? If it grows, what gets added to it? When a program executes in a high context, one of three things can happen (and all three things can happen during different parts of the execution). Firstly, the program can branch and pop the appropriate number of entries off the stack. Secondly, the program can pop some number of entries off the stack without branching. Thirdly, the program can push elements onto the stack. In the first two cases, the bottom of the stack will remain unchanged between the beginning and the end of the execution. In the third case, there is still some part at the bottom of the stack that remains unchanged (this may however be empty) and the top of the stack will contain only values labeled at or above the high *pc*-label. Relation $\blacktriangleleft_{\mathcal{A}}$ in Figure 7b captures all three cases (Definition 5 in Section 5).

3.5 Big-Step Semantics

To conclude this section, we make a final note on a decision related to the semantic model we take to obtain proof clarity and simplicity.

In this paper, we opt for a big-step operational semantics for (Sec)Wasm, in contrast to previous work using a small-step operational semantics [22], due to two principal reasons. Firstly, our goal is to provide an IFC system that is mostly static and, therefore, we do not find the choice of semantics to be crucial, as long as it remains faithful to the Wasm specification. Secondly, our IFC system aims to provide end-to-end noninterference for full program executions. In this setting, big-step semantics naturally accommodates clean proofs of noninterference for Wasm's structured control flow primitives.

4 SecWasm

This section presents the technical details of SecWasm, our information flow-aware variant of Wasm. Recall we focus on WebAssembly 1.0 [47]. Consequently, we disregard language extensions in the current version [50]. However, to the best of our knowledge, the extensions do not fundamentally alter Wasm in a way that could not be accommodated in SecWasm.

4.1 Syntax

As already discussed in the previous section, SecWasm extends several of Wasm syntactic constructs with security levels, all highlighted in Figure 8. We append a security label ℓ to each value type, and augment all types t in Wasm to labeled types τ in SecWasm. Further, we annotate function types *ft* with a security label ℓ specifying an upper bound on the information that may flow into the execution of a function. As mentioned in Section 3, instructions for reading from/writing

(security labels)	ℓ	::=	$\texttt{L} \mid \texttt{H} \mid \ldots$
(labeled types)	τ	::=	$t\langle\ \ell\ \rangle$
(global types)	gt	::=	$\mathsf{mut}^{?}\ \tau$
(function types)	ft	::=	$\tau^* \xrightarrow{\ell} \tau^*$
(block types)	bt	::=	$\tau^* \to \tau^*$
(memory instructions)	mem	::=	$t.\mathbf{load}\ \ \ell \mid t.\mathbf{store}\ \ \ell$
(admin instructions)	$admin$	::=	$\mathbf{trap}$

Fig. 8: SecWasm's extensions over Wasm syntax.

to memory also carry a security label ℓ. We omit alignment immediates for these instructions as they do not affect the semantics [50]. As seen in Section 2, administrative instructions are an artifact of small-step semantics. Due to the big-step semantics paradigm we employ, all administrative operators except for **trap** become irrelevant in SecWasm.

As our extensions are only related to information-flow, we do not explicitly distinguish between SecWasm and Wasm when we discuss about the syntax and semantics the two systems share. We use SecWasm only when we refer to the information-flow extensions to Wasm.

4.2 Semantics

Since our IFC enforcement is mostly static, this subsection provides mainly a glimpse into (Sec)Wasm's semantic behavior.

Notation If a is a sequence or stack of items, then we use notation $a[i]$ to denote the i:th element of the stack (counting from top and starting from 0), $a[i:]$ to denote all elements from $a[i]$ through the end of a, and $a[i:j]$ to denote all elements from $a[i]$ to $a[j]$ inclusive (the empty sequence is $j < i$ and $a[i:\infty]$ is equivalent to $a[i:]$). Furthermore, we write $a[i:j \to k^*]$ to denote the sequence in a with all data at indices between (inclusive) i and j replaced by the sequence of values k^*. We use :: as a stack entry separator. Note in SecWasm, we represent the top of the stack on the left, i.e., $a[0] :: a[1:]$, unlike in pure Wasm, where it is denoted on the right.

By e^n we denote a sequence of length n with all free variables in e replaced by x_i for each $i \in [0, n-1]$.

Following Wasm, we make heavy use of record-like syntactic constructs in SecWasm. A grammatical category consisting of records is declared, e.g., as $R ::= \{\mathsf{key}_1\ n, \mathsf{key}_2\ expr\}$ and if $r \in R$ then $r = \{\mathsf{key}_1\ n, \mathsf{key}_2\ expr\}$ for some number n and expression $expr$, and $r.\mathsf{key}_1 = n$. Furthermore, we use syntax $r\{\mathsf{key}_1\ 0\}$ to denote a record that is like r except "field" key_1 now has value 0.

Evaluation Judgment As discussed in Section 3, we employ a big-step semantics paradigm due to its cleaner representation and ease of reasoning. As such, we have a big-step evaluation judgment $\langle\!\langle \sigma, S, expr \rangle\!\rangle \Downarrow \langle\!\langle \sigma', S', \theta \rangle\!\rangle$ relating an initial configuration to a final configuration. In the initial configuration, a sequence of instructions $expr$ is executed in current state S by interacting with the operand

$$\begin{array}{lll}
\text{(values)} & v & ::= t.\mathsf{const}\ k \\
\text{(addresses)} & a & ::= 0 \mid 1 \mid 2 \mid \ldots \\
\text{(store)} & S & ::= \{\mathsf{funcs}\ func^*_{inst}, \mathsf{tables}\ table^*_{inst}, \mathsf{globals}\ global^*_{inst}, \mathsf{mems}\ mem^*_{inst}\} \\
\text{(function instances)} & func_{inst} & ::= \{\mathsf{type}\ i, \mathsf{module}\ module_{inst}, \mathsf{code}\ func\} \\
\text{(memory instances)} & mem_{inst} & ::= \{\mathsf{data}\ (byte, \ell)^*, \mathsf{max}\ k^?\} \\
\text{(operand stack)} & \sigma & ::= \varepsilon \mid v :: \sigma \mid L_k :: \sigma \mid \mathbf{frame}_k\{frame\} :: \sigma \\
\text{(frames)} & frame & ::= \{\mathsf{locals}\ v^*, \mathsf{module}\ module_{inst}\}
\end{array}$$

Expression evaluation: $\boxed{\langle\!\langle \sigma, S, expr \rangle\!\rangle \Downarrow \langle\!\langle \sigma', S', \theta \rangle\!\rangle}$

E-LOAD

$$\frac{\begin{array}{c} j = i + S.\mathsf{mem.offset} \\ j + |t|/8 \leq S.\mathsf{mem.data} \qquad S.\mathsf{mem}[j : j + |t|/8] = (b, \ell)^* \qquad bytes_t(n) = b^* \qquad \bigsqcup \ell \sqsubseteq \ell_m \end{array}}{\langle\!\langle \mathsf{i32}.\mathbf{const}\ i :: \sigma, S, t.\mathbf{load}\ \ell_m \rangle\!\rangle \Downarrow \langle\!\langle t.\mathbf{const}\ n :: \sigma, S, \textit{no-br} \rangle\!\rangle}$$

E-STORE

$$\frac{\begin{array}{c} j = i + S.\mathsf{mem.offset} \\ j + |t|/8 \leq S.\mathsf{mem.data} \qquad bytes_t(n) = b^* \qquad S' = S.\mathsf{mem}[j : j + |t|/8 \mapsto (b, \ell_m)^*] \end{array}}{\langle\!\langle t.\mathbf{const}\ n :: \mathsf{i32}.\mathbf{const}\ i :: \sigma, S, t.\mathbf{store}\ \ell_m \rangle\!\rangle \Downarrow \langle\!\langle \sigma, S', \textit{no-br} \rangle\!\rangle}$$

E-MEMORY-GROW

$$\frac{\begin{array}{c} \sigma|_F[0].\mathsf{module.memaddrs}[0] = a \qquad S.\mathsf{mems}[a] = m \qquad sz = |m.\mathsf{data}|/64\ \mathrm{Ki} \qquad len = k + sz \\ len \leq 2^{16} \qquad (m.\mathsf{max} = null \ \vee\ len \leq m.\mathsf{max}) \qquad S' = S.\mathsf{mems}[a][sz : len \to (0, \mathtt{L})] \end{array}}{\langle\!\langle \mathsf{i32}.\mathbf{const}\ k :: \sigma, S, \mathbf{memory.grow} \rangle\!\rangle \Downarrow \langle\!\langle \mathsf{i32}.\mathbf{const}\ sz :: \sigma, S', \textit{no-br} \rangle\!\rangle}$$

E-BLOCK

$$\frac{\begin{array}{c} \langle\!\langle v_1^n :: L_m :: \sigma_{init}, S, expr \rangle\!\rangle \Downarrow \langle\!\langle \sigma, S', \theta \rangle\!\rangle \\ \theta \neq \textit{no-br} \Rightarrow \sigma_{\mathit{fin}} = \sigma \qquad \theta = \textit{no-br} \Rightarrow (\sigma = \sigma' :: L_m^0 :: \sigma'' \wedge \sigma_{\mathit{fin}} = \sigma' :: \sigma'') \end{array}}{\langle\!\langle v_1^n :: \sigma_{init}, S, \mathbf{block}\ (\tau_1^n \to \tau_2^m)\ expr\ \mathbf{end} \rangle\!\rangle \Downarrow \langle\!\langle \sigma_{\mathit{fin}}, S', \mathsf{pred}(\theta) \rangle\!\rangle}$$

E-LOOP-EVAL

$$\frac{\langle\!\langle v_1^n :: L_n :: \sigma, S, expr \rangle\!\rangle \Downarrow \langle\!\langle \sigma', S', 0 \rangle\!\rangle \qquad \langle\!\langle \sigma', S', \mathbf{loop}\ (\tau_1^n \to \tau_2^m)\ expr\ \mathbf{end} \rangle\!\rangle \Downarrow \langle\!\langle \sigma'', S'', \theta \rangle\!\rangle}{\langle\!\langle v_1^n :: \sigma, S, \mathbf{loop}\ (\tau_1^n \to \tau_2^m)\ expr\ \mathbf{end} \rangle\!\rangle \Downarrow \langle\!\langle \sigma'', S'', \theta \rangle\!\rangle}$$

E-BR-IF-JUMP

$$\frac{}{\langle\!\langle \mathsf{i32}.\mathbf{const}\ k+1 :: v^n :: \sigma_0 :: L_n^{i-1} :: \sigma, S, \mathbf{br\_if}\ i \rangle\!\rangle \Downarrow \langle\!\langle v^n :: \sigma, S, i \rangle\!\rangle}$$

E-BR-IF-NO-JUMP

$$\frac{}{\langle\!\langle \mathsf{i32}.\mathbf{const}\ 0 :: \sigma, S, \mathbf{br\_if}\ i \rangle\!\rangle \Downarrow \langle\!\langle \sigma, S, \textit{no-br} \rangle\!\rangle}$$

E-RETURN

$$\frac{}{\langle\!\langle v^n :: \sigma :: F_n, S, \mathbf{return} \rangle\!\rangle \Downarrow \langle\!\langle v^n :: F_n, S, \textit{return} \rangle\!\rangle}$$

E-CALL

$$\frac{\begin{array}{c} f = S.\mathsf{funcs}[i] \qquad f.\mathsf{type} = \tau_1^n \overset{\ell}{\to} \tau_2^m \qquad f.\mathsf{code.locals} = \tau^p \qquad f.\mathsf{code.body} = expr \\ F_m = \{\mathsf{locals}\ v_1^n : (t.\mathbf{const}\ 0)^p, \mathsf{module}\ f.\mathsf{module}\} \qquad \langle\!\langle F_m, S, expr \rangle\!\rangle \Downarrow \langle\!\langle v_2^m :: F_m, S', \theta \rangle\!\rangle \end{array}}{\langle\!\langle v_1^n :: \sigma, S, \mathbf{call}\ i \rangle\!\rangle \Downarrow \langle\!\langle v_2^m :: \sigma, S', \textit{no-br} \rangle\!\rangle}$$

E-SEQ-JUMP

$$\frac{\langle\!\langle \sigma_0, S_0, expr_0 \rangle\!\rangle \Downarrow \langle\!\langle \sigma_1, S_1, \theta \rangle\!\rangle \qquad \theta \neq \textit{no-br}}{\langle\!\langle \sigma_0, S_0, expr_0; expr_1 \rangle\!\rangle \Downarrow \langle\!\langle \sigma_1, S_1, \theta \rangle\!\rangle}$$

E-SEQ

$$\frac{\begin{array}{c} \langle\!\langle \sigma_0, S_0, expr_0 \rangle\!\rangle \Downarrow \langle\!\langle \sigma_1, S_1, \textit{no-br} \rangle\!\rangle \\ \langle\!\langle \sigma_1, S_1, expr_1 \rangle\!\rangle \Downarrow \langle\!\langle \sigma_2, S_2, \theta \rangle\!\rangle \end{array}}{\langle\!\langle \sigma_0, S_0, expr_0; expr_1 \rangle\!\rangle \Downarrow \langle\!\langle \sigma_2, S_2, \theta \rangle\!\rangle}$$

Fig. 9: SecWasm selected evaluation rules. Security extensions are highlighted.

stack σ, leading to the final configuration containing the updated state S' and operand stack σ'. The essence of this paradigm is the third component θ of a final configuration. θ evaluates to either a natural number j denoting a branch out of j contexts (blocks, loops, or conditionals), *no-br* if there was no jump, or *return* if a **return** instruction executed. θ allows to do away with the administrative instructions in Wasm. More on this in the next paragraph when we discuss selected evaluation rules.

Metavariable S represents the store or the global state and comprises of instances for all functions, globals, tables, and memories that have been allocated. Just like in pure Wasm, operand stack σ contains three types of entries: values, labels, and frames. In SecWasm, we diverge slightly from Wasm by denoting branch target labels as L_n instead of $\textbf{label}_n\{expr\}$, as in SecWasm we do not need to keep track of the continuation expression *expr*. As a simplifying choice, we also use the syntax $\sigma :: L_n^{i-1} :: \sigma'$ to represent the case where L_n is the i:th label (counting from top and starting from 0) on the compound stack $\sigma :: L_n :: \sigma'$. Frames remain as defined in Wasm, $\textbf{frame}_n\{frame\}$, with *frame* keeping track of the values for the function's local variables.

Another point of divergence from Wasm is that in SecWasm there is only one frame on the operand stack at any given time. The reason for this change is that it simplifies our formalization. Thus, instead of having an operand stack containing several frames, in SecWasm every function call creates another (sub-)stack, where its corresponding frame is on the bottom. This is in line with function behavior in WebAssembly, as jumps from inside a function are either branching from within nested blocks, giving control at the end of the corresponding block, or **return**s, giving control back to the caller function. This will become more obvious when discussing rules E-CALL-*.

Similar to Wasm, abnormal termination of a program results in a trap, denoted $\ll\sigma, S, expr\gg \Downarrow \textbf{trap}$. When a trap occurs, the computation is aborted and no further modifications to the state can be made. In SecWasm, the execution of an instruction traps under the same conditions as in Wasm, but failure to satisfy the additional security checks also leads to a trap. Thus, SecWasm introduces additional rules for handling the error cases which result in a trap due to the IFC-checks. These rules are presented in the technical report [6].

Selected Evaluation Rules Figure 9 depicts the most important evaluation rules, while the full set of rules is presented in the technical report [6]. Since we opt for a mostly static enforcement, note only few semantic rules carry security checks.

The intuition for the memory access rules was given in Section 3, so we do not discuss the rules in detail here. However, recall Examples 1 and 2 and note premise $\bigsqcup \ell \sqsubseteq \ell_m$ in rule E-LOAD ensuring all security levels ℓ of memory locations read from are below the immediate label ℓ_m for the **load** instruction. Due to this check, in SecWasm the execution of Example 1 will trap, while the execution of Example 2 will succeed. Further, recall Example 3 and note that rule E-STORE updates the security levels of the memory locations written into with no additional checks.

Before we discuss the rules for achieving structured control flow, few things are worth mentioning. First, recall that branching can only happen from within the block constructs **block**, **loop**, and **if**. Second, the end of every such block is a valid branch target for code executing inside the block, with the exception of loops where the target can also be at the start of the loop. Finally, recall θ specifies how far out of a series of nested blocks to jump. We further introduce the notion of predecessor of θ ($\mathsf{pred}(\theta)$) specifying how to update θ when we exit a block: $\mathsf{pred}(\mathit{no\text{-}br}) = \mathsf{pred}(0) = \mathit{no\text{-}br}$, $\mathsf{pred}(j+1) = j$, $\mathsf{pred}(\mathit{return}) = \mathit{return}$.

When entering a **block** of type $\tau_1^n \to \tau_2^m$ and body *expr*, label L_m is added in between the top n values v_1^n of the operand stack corresponding to the block's input arguments and the rest of the stack. Exiting a block can happen either by trapping (rule E-BLOCK-TRAP), by jumping (when a branch/return instruction is executed inside the block), or by reaching its end without a jump. Rule E-BLOCK distinguishes between the latter two cases by inspecting marker θ. If no jump occurred ($\theta = \mathit{no\text{-}br}$), we remove the label L_m from the operand stack and return the result $\sigma' :: \sigma''$. Otherwise, we return the operand stack as is, since the stack unwinding has been dealt with already by the jumping instruction (See below rule E-BR-IF-JUMP.) Finally, function pred adjusts θ to account for the fact that a block has been exited.

Consider again Example 7 when $x \neq 0$ and the instruction on line 8 is about to be executed. **br** 1 unconditionally jumps out of the two blocks and gives control at the end of instruction on line 11. θ is set to 1 after executing line 8 and exiting block \$1 updates it to $\mathsf{pred}(1) = 0$ (rule E-BLOCK). Since $\theta \neq \mathit{no\text{-}br}$, all remaining instructions in block \$0 will be ignored (rule E-SEQ-JUMP). Reaching the end of block \$0 updates θ again to $\mathsf{pred}(0) = \mathit{no\text{-}br}$. If present, executing all subsequent instructions would continue according to rule E-SEQ until the next branching or function return.

loop and **if** statements constitute **block**s with slightly specialized rules to reflect their different function. This can also be seen in the semantic behavior of pure Wasm, where **if**s and **loop**s reduce in one step to a **block** [22]. For this reason we only present rules E-LOOP-SKIP (for leaving a loop) and E-IF in the technical report [6], as they differ only slightly from rule E-BLOCK. What differs is that **if** statements choose the expression to execute based on the value on top of the operand stack, while E-LOOP-SKIP requires θ to be different than 0, as $\theta = 0$ restarts the loop (rule E-LOOP-EVAL). Note from rule E-LOOP-EVAL another perk of Wasm, namely **loop** blocks are evaluated at least once.

A conditional branch **br_if** i executes when the value on top of the operand stack is different than 0 (rule E-BR-IF-JUMP). In this case, Wasm requires the top of the stack to contain at least n other values, as illustrated by the index of the i:th label L_n^{i-1} on the input stack. Recall the index specifies the number of values expected by the branch target. Next, the rule drops everything between the top $n+1$ entries on the stack down to and including label L_n^{i-1} and finishes with $\theta = i$. If the top value of the operand stack is 0, then the conditional branch does not execute (rule E-BR-IF-NO-JUMP), and the computation proceeds sequentially, finishing with $\theta = \mathit{no}\text{-}\mathit{br}$. Unconditional branching **br** i (rule E-BR) works in a similar way as executing conditional branching.

When a function is **call**ed (rules E-CALL-*), we create an empty operand stack and push on it a frame instantiated with values v_1^n for the function arguments and initial values 0 for the function's local variables. When returning from a function, we only retain the return values, discarding everything else, including the frame. Note in Wasm, the frame is popped off when executing a **return**, but in SecWasm it is not (rule E-RETURN).

Finally, rules E-SEQ-* distinguish between the cases when a jump occurred, i.e., $\theta \neq \mathit{no}\text{-}\mathit{br}$ in rule E-SEQ-JUMP, and when the execution proceeds sequentially in rule E-SEQ. In the former case, rule E-SEQ-JUMP simply ignores the subsequent instructions until θ becomes $\mathit{no}\text{-}\mathit{br}$. And the block rules ensure θ indeed decreases to $\mathit{no}\text{-}\mathit{br}$, by computing its predecessor every time a block is exited. Thus, either the same number of blocks have been exited as the initial value of $\theta + 1$, or all instructions after a **return** statement have been ignored.

4.3 Security Type System

As our enforcement is mostly static, SecWasm's type system is heavily populated with security checks. Before discussing the type system, we first give an intuition for the constructs SecWasm uses to track the information flows, and then briefly discuss the typing judgment.

Tracking Flows—an Intuition As the bedrock for static IFC in Wasm, SecWasm's type system tracks both explicit and implicit information flows. For tracking explicit flows, we assign a security label to each value in the operand stack via a *type stack* st denoting a stack of labeled types. As discussed in Section 3.3, for tracking implicit flows we use a stack of pc labels, with a label entry for every block context. We then combine the pc stack with the type stack in a *stack-of-stacks* γ with entries $\langle st, pc\rangle$. Upon entering a block, γ is augmented with a new pair $\langle st, pc\rangle$, with st denoting the input stack for the block, and pc the initial program counter label for the block's execution. The security labels in γ may get upgraded, and after leaving a block, the top two entries are merged.

Typing Judgments The type system assumes a typing security context C containing e.g., the type of functions and local variables. C is defined as in Wasm, but where value types t have been adorned with labels to labeled types τ.

Previous presentations of Wasm [22] depict the type system using a judgment of the form $C \vdash \mathit{expr} : t^n \to t^m$ that only says how expr affects the top elements on the stack and leaves the rest to a subtyping-like rule. Instead, we use a more explicit judgment form passing the entire γ around while updating its program counters: $\gamma, C \vdash \mathit{expr} \dashv \gamma'$. The judgment reads as follows: Assuming input type stack $\gamma.\mathsf{fst}$ and security context C, expr produces (possibly) updated output type stack $\gamma'.\mathsf{fst}$. For $\gamma = \langle st_0, pc_0\rangle :: \ldots :: \langle st_n, pc_n\rangle$, $\gamma.\mathsf{fst}$ denotes the stack formed by the first elements of each entry in γ, i.e., $\gamma.\mathsf{fst} \triangleq st_0 :: \ldots :: st_n$.

We extend the type system with a simple subtyping judgment for types to capture when a type is less sensitive than another and write $\tau \sqsubseteq \tau'$ whenever the

(Security contexts)	C	$::= \{\mathsf{globals}\ (\mathsf{mut}^{?}\ \tau)^{*}, \mathsf{locals}\ \tau^{*}, \mathsf{return}\ (\tau^{*})^{?}, \mathsf{labels}\ (\tau^{*})^{*}, \ldots\}$
(Security-labeled type stack)	st	$::= \varepsilon \mid \tau :: st$
(Stack-of-stacks)	γ	$::= \varepsilon \mid (st, pc) :: \gamma$

Expression typing: $\boxed{\gamma, C \vdash expr \dashv \gamma'}$

$$\textsc{T-unreachable}\quad \frac{}{\gamma, C \vdash \textbf{unreachable} \dashv \gamma}$$

$$\textsc{T-load}\quad \frac{C.\mathsf{mem} = n \qquad \ell_v = \ell_a \sqcup \ell \sqcup pc}{\langle \mathsf{i32}\langle \ell_a \rangle :: st,\ pc \rangle :: \gamma, C \vdash t.\textbf{load}\ \ell \dashv \langle t\langle \ell_v \rangle :: st,\ pc \rangle :: \gamma}$$

$$\textsc{T-store}\quad \frac{C.\mathsf{mem} = n \qquad pc \sqcup \ell_a \sqcup \ell_v \sqsubseteq \ell}{\langle t\langle \ell_v \rangle :: \mathsf{i32}\langle \ell_a \rangle :: st,\ pc \rangle :: \gamma, C \vdash t.\textbf{store}\ \ell \dashv \langle st,\ pc \rangle :: \gamma}$$

$$\textsc{T-memory-grow}\quad \frac{C.\mathsf{mem} = n}{\langle \mathsf{i32}\langle \mathtt{L} \rangle :: st,\ \mathtt{L} \rangle :: \gamma, C \vdash \textbf{memory.grow} \dashv \langle \mathsf{i32}\langle \mathtt{L} \rangle :: st,\ \mathtt{L} \rangle :: \gamma}$$

$$\textsc{T-block}\quad \frac{\langle \tau_1^n,\ pc \rangle :: \langle st,\ pc \rangle :: \gamma, \mathsf{label}(\tau_2^m) : C \vdash expr \dashv \langle \tau_2^m,\ pc' \rangle :: \langle st',\ pc'' \rangle :: \gamma'}{\langle \tau_1^n :: st,\ pc \rangle :: \gamma, C \vdash \textbf{block}\ (\tau_1^n \to \tau_2^m)\ expr\ \textbf{end} \dashv \langle \tau_2^m :: st',\ pc \sqcup pc'' \rangle :: \gamma'}$$

$$\textsc{T-loop}\quad \frac{\begin{array}{c} pc \sqsubseteq pc' \qquad \gamma \sqsubseteq \gamma' \qquad pc \sqsubseteq pc'' \qquad st \sqsubseteq st' \\ \langle \tau_1^n,\ pc' \rangle :: \langle st',\ pc'' \rangle :: \gamma', \mathsf{label}(\tau_1^n) : C \vdash expr \dashv \langle \tau_2^m,\ pc' \rangle :: \langle st',\ pc'' \rangle :: \gamma' \end{array}}{\langle \tau_1^n :: st,\ pc \rangle :: \gamma, C \vdash \textbf{loop}\ (\tau_1^n \to \tau_2^m)\ expr\ \textbf{end} \dashv \langle \tau_2^m :: st',\ pc \sqcup pc'' \rangle :: \gamma'}$$

$$\textsc{T-br-if}\quad \frac{C.\mathsf{labels}[i] = st \qquad \gamma \sqsubseteq \gamma' \qquad pc \sqcup \ell \sqsubseteq st \qquad \gamma^* = \mathtt{lift}_{\ell \sqcup pc}(\langle st :: st', pc \rangle :: \gamma'[0 : i-1])}{\langle \mathsf{i32}\langle \ell \rangle :: st :: st',\ pc \rangle :: \gamma, C \vdash \textbf{br\_if}\ i \dashv \gamma^* :: \gamma'[i :]}$$

$$\textsc{T-return}\quad \frac{\begin{array}{c} C.\mathsf{return} = st \qquad \gamma \sqsubseteq \gamma' \qquad pc \sqsubseteq st \\ \gamma'' = \mathtt{lift}_{pc}(\langle st'', \ell \rangle :: \gamma') \end{array}}{\langle st :: st',\ pc \rangle :: \gamma, C \vdash \textbf{return} \dashv \gamma''}$$

$$\textsc{T-call}\quad \frac{C.\mathsf{funcs}[i] = f : \tau_1^n \xrightarrow{\ell} \tau_2^m \qquad pc \sqsubseteq \ell}{\langle \tau_1^n :: st,\ pc \rangle :: \gamma, C \vdash \textbf{call}\ i \dashv \langle \tau_2^m :: st,\ pc \rangle :: \gamma}$$

$$\textsc{T-call-indirect}\quad \frac{pc \sqcup \ell \sqsubseteq \ell_f}{\langle \mathsf{i32}\langle \ell \rangle :: \tau_1^n :: st,\ pc \rangle :: \gamma, C \vdash \textbf{call\_indirect}\ \tau_1^n \xrightarrow{\ell_f} \tau_2^m \dashv \langle \tau_2^m :: st,\ pc \rangle :: \gamma}$$

Fig. 10: SecWasm type system (Selected rules). Security extensions and static checks are highlighted.

label of τ can flow to the label of τ'. We further extend this notion to sequences of labeled types as $st \sqsubseteq st'$ if st and st' are of the same length and $\tau_i \sqsubseteq \tau_i'$ for $\tau_i = st[i]$ and $\tau_i' = st'[i]$, respectively.

Selected Typing Rules In the following, we discuss the most interesting rules of the type system, depicted in Figure 10. The full set of rules is presented in the technical report [6].

First, note that abuses of non-termination channel such as in snippet t.**load** H; **br_if** 0; **unreachable** are outside the scope of this work, as we further focus on enforcing termination-insensitive noninterference. Thus, we add no restrictions on the program context in rule T-UNREACHABLE.

An intuition for the memory access instructions was given in Section 3. Here, we reiterate that static security checks are employed only when writing to the memory ($pc \sqcup \ell_a \sqcup \ell_v \sqsubseteq \ell$ in T-STORE), as the semantics are responsible for the dynamic security checks when reading. Finally, **memory.grow** executes in a public context and only if the amount to extend the memory with is also public.

Typing the **block** instruction (rule T-BLOCK) requires the current type stack to contain at least n labeled types, corresponding to the block type. Since we enter a new block, we split the arguments off and push pair $\langle \tau_1^n, pc \rangle$ containing the n labeled types and the same program counter pc on the stack-of-stacks $\langle st, pc \rangle :: \gamma$. We also push τ_2^m on the label-stack $C.\mathsf{labels}$ in context C to denote the branch target at the end of the block ($\mathsf{label}(\tau_2^m) : C$). The sequence of instructions *expr* is required to produce m correctly typed output values and a new stack of stacks $\langle st', pc'' \rangle :: \gamma'$ possibly with higher labels. Finally, on the output stack-of-stacks, τ_2^m is merged with st'.

Recall **if** and **loop** are just special types of **block**s. As a consequence, rules T-IF and T-LOOP only bear minor differences to rule T-BLOCK. For the former, inner expressions $expr_1$ and $expr_2$ are type-checked under a program counter *tainted* by the information flow from the condition operand, and for the latter, the labels of type stacks and program counter need to be in a fixed-point over the loop.

In rule T-BR-IF, all types on the stack-of-stacks $\langle st, pc \rangle :: \gamma$ until and including the i:th+1 entry are tainted by label ℓ of the top element on the input stack deciding whether a branch will happen, as illustrated in Example 6. (This is represented by operator `lift` upgrading all security levels present in its argument.) Furthermore, we require $pc \sqcup \ell \sqsubseteq C.\mathsf{labels}[i]$ to avoid implicit flows. This rule is important because it rejects leaky programs like the one in Example 8 that copies the truth-value of local variable $y_{\mathtt{H}}$ to local variable x_{L} by skipping all the way to the end with **br_if** 1.

Example 8.

```
block
  block
    i32.const 0
    local.get y_H
    br_if 1
  end
  drop
  i32.const 1
end
local.set x_L
```

All other jumping rules entail a similar taint propagation. In rule T-RETURN, for example, the entire stack-of-stacks is tainted by the function program counter. Note that premise $pc \sqsubseteq st$ in the jumping rules is synthetic and we resort to using it as it considerably simplifies the proofs.

Rule T-CALL is standard for function calls in IFC type systems. The input type stack is required to be a subtype of the input type stack for the caller function, the function program counter label ℓ needs to be at least as high as current callee pc, and the output type stack of the function needs to be a subtype of the expected output type stack.

T-CALL-INDIRECT works in almost the same way as rule T-CALL, with the difference that indirect calls require a 32-bit integer labeled ℓ on top of the input stack acting as the function pointer and thus the function also needs to check ℓ flows to the function program counter ℓ_f.

5 Security Properties

This section presents the security properties enforced by SecWasm. All proofs are manual and presented in the technical report [6], a mechanization thereof being left for future work.

We begin by stating two well-formedness properties for operand stacks $C \vdash \sigma$ and stores $C \vdash S$, specifying that local and global variables are well-typed in σ and S, respectively, with respect to the types declared in context C.

Definition 1 (Context-Stack Well-Formedness). *Operand stack σ is well-formed with respect to context C, denoted $C \vdash \sigma$, if:*

1. *For all i in the domain of $C.\mathsf{labels}$ there exists some σ_0, σ_1, and m such that $\sigma = \sigma_0 :: L^i_m :: \sigma_1$ and $C.\mathsf{labels}[i] = \tau^m$ for some τ^m.*
2. *$C.\mathsf{return} = \tau^m$ for some m and $\sigma|_F[0] = F_m$, for the bottom frame F_m and $F_m.\mathsf{locals}$ is well typed with respect to $C.\mathsf{locals}$.*

Definition 2 (Context-Store Well-Formedness). *Store S is well-formed with respect to context C, denoted $C \vdash S$, if:*

1. *For every function f in $S.\mathsf{funcs}$ we have $C \vdash f$.*
2. *For every variable in $C.\mathsf{globals}$ there is a corresponding well-typed entry in $S.\mathsf{globals}$.*

Next, we state what it means for an operand stack and labeled type stacks to be in agreement. (Recall Figure 7a.)

Definition 3 (Operand Stack and Type Stack Agreement). *Given operand stack σ and type stack st, we define σ agreement with st (denoted $st \Vdash \sigma$) inductively as:*

$$\frac{}{[] \Vdash \varepsilon} \qquad \frac{st \Vdash \sigma}{t\langle\ell\rangle :: st \Vdash t.\mathbf{const}\ k :: \sigma} \qquad \frac{st \Vdash \sigma}{st \Vdash L :: \sigma} \qquad \frac{st \Vdash \sigma}{st \Vdash F :: \sigma}.$$

Now, we can define what it means for two operand stacks to be equivalent with respect to the attacker, i.e., relations $\sim_{\mathcal{A}}$ and $\blacktriangleleft_{\mathcal{A}}$, as discussed in Section 3. Recall security label $\mathcal{A}$ simply captures the level at or below which the attacker can read information.

Definition 4 (Operand Stack and Type Stack Agreement Equivalence). *For two operand stacks σ_0 and σ_1 and type stacks st_0 and st_1 such that $st_i \Vdash \sigma_i$, we define operand stack equivalence $st_0 \Vdash \sigma_0 \sim^C_{\mathcal{A}} st_1 \Vdash \sigma_1$ inductively as:*

$$\frac{}{[] \Vdash \varepsilon \sim^C_{\mathcal{A}} [] \Vdash \varepsilon} \qquad \frac{st_0 \Vdash \sigma_0 \sim^C_{\mathcal{A}} st_1 \Vdash \sigma_1 \qquad \ell_0 \sqsubseteq \mathcal{A} \wedge \ell_1 \sqsubseteq \mathcal{A} \Rightarrow v_0 = v_1}{t\langle \ell_0 \rangle :: st_0 \Vdash v_0 :: \sigma_0 \sim^C_{\mathcal{A}} t\langle \ell_1 \rangle :: st_1 \Vdash v_1 :: \sigma_1}$$

$$\frac{st_0 \Vdash \sigma_0 \sim^C_{\mathcal{A}} st_1 \Vdash \sigma_1 \qquad F \sim^C_{\mathcal{A}} F'}{st_0 \Vdash F :: \sigma_0 \sim^C_{\mathcal{A}} st_1 \Vdash F' :: \sigma_1} \qquad \frac{st_0 \Vdash \sigma_0 \sim^C_{\mathcal{A}} st_1 \Vdash \sigma_1}{st_0 \Vdash L :: \sigma_0 \sim^C_{\mathcal{A}} st_1 \Vdash L :: \sigma_1}.$$

The two type stacks st_0 and st_1 must have the same *shape*, but may differ in their security labels. This allows us to relate prefixes of stacks before and after program execution (when security labels may have been upgraded due to a branch). In other words, this part of the definition does not come into effect when considering a “traditional” noninterference theorem statement.

Ideally, when proving noninterference one would show that if two configurations, including stacks and memories, are $\mathcal{A}$-equivalent then the output configurations that result after executing the same program on both these configurations are also $\mathcal{A}$-equivalent. However, this property cannot easily be extended to be inductive and instead a *confinement* lemma is required. This lemma relates the configurations before and after a single execution in a high context. Specifically, it usually says that when you execute a well-typed program in a high context it only alters high data. However, this statement is not sufficient in SecWasm, as we also have to specify what happens to the operand stack during this execution.

And this is how we define ordered equivalence $\blacktriangleleft_{\mathcal{A}}$, by introducing judgment $\gamma \Vdash \sigma \blacktriangleleft^C_{\mathcal{A}} \gamma' \Vdash \sigma'$ stating that stack σ' is the result of executing a high (w.r.t. the attacker-label $\mathcal{A}$) program that starts off with σ. To prove σ and σ' are related in this way one needs to prove there is some common $\mathcal{A}$-equivalent bottom of the two stacks (that may be empty) and that all elements on top of this bottom part of σ' are labeled high in γ'.

Definition 5 (Operand Stack and Stack-of-Stacks Agreement Ordered Equivalence).

$$\frac{\begin{array}{c}\gamma \Vdash \sigma_t :: \sigma_b \qquad \gamma' \Vdash \sigma'_t :: \sigma'_b \qquad \gamma.\mathsf{fst} = st_t :: st_b \\ \gamma'.\mathsf{fst} = st'_t :: st'_b \qquad st_b \sqsubseteq st'_b \qquad \mathtt{high}(st'_t) \qquad st_b \Vdash \sigma_b \sim^C_{\mathcal{A}} st'_b \Vdash \sigma'_b\end{array}}{\gamma \Vdash \sigma_t :: \sigma_b \blacktriangleleft^C_{\mathcal{A}} \gamma' \Vdash \sigma'_t :: \sigma'_b}$$

Note the *pc*s are not used in the ordered equivalence, although they are part of γ. The reason for this is that in our proofs we only require the structure of $\gamma.\mathsf{fst}$ given by γ.

Recall from the discussion in Section 3 that the classical memory equivalence is not strong enough for our setting, so we use an ordered-equivalence relation $\blacktriangleleft_{\mathcal{A}}$ which says that two linear memories m and m' are $\blacktriangleleft_{\mathcal{A}}$-ordered equivalent if m has strictly more high-labeled indices and all the low-labeled indices are the same between m and m'.

Definition 6 ($\mathcal{A}$-Ordered Memory Equivalence). *Two memories m_0 and m_1 are $\mathcal{A}$-ordered equivalent (denoted $m_0 \blacktriangleleft_{\mathcal{A}} m_1$) iff $\forall l.\ m_1(l) = (k, \ell) \wedge \ell \sqsubseteq \mathcal{A} \Rightarrow m_0(l) = (k, \ell)$ and $\forall l.\ m_1(l) = (k_1, \ell_1) \wedge \ell_1 \not\sqsubseteq \mathcal{A} \Rightarrow m_0(l) = (k_0, \ell_0) \wedge \ell_1 \not\sqsubseteq \ell_0$.*

Further, we also need to consider what happens to the linear memory, global and local variables, i.e., the state of the program. Fortunately, the flow-insensitive nature of the global and local variables means that these will just be $\mathcal{A}$-equivalent before and after execution.

Definition 7 ($\mathcal{A}$-Ordered Store Equivalence). *Two stores S_0 and S_1 are $\mathcal{A}$-ordered equivalent given security context C:*

$$S_0 \blacktriangleleft^{C}_{\mathcal{A}} S_1 \text{ iff } \begin{cases} S_0.\mathsf{funcs} = S_1.\mathsf{funcs} \\ S_0.\mathsf{tables} = S_1.\mathsf{tables} \\ S_0.\mathsf{globals} \sim^{C}_{\mathcal{A}} S_1.\mathsf{globals} \\ S_0.\mathsf{mems} \blacktriangleleft^{C}_{\mathcal{A}} S_1.\mathsf{mems}. \end{cases}$$

Confinement Usually, these definitions are sufficient for stating confinement. Yet, in SecWasm we need to deal with an unwinding stack too. Ideally, confinement would be that given $\gamma, C \vdash \mathit{expr} \dashv \gamma'$ where $\gamma[0].\mathsf{snd} \not\sqsubseteq \mathcal{A}$ and $\langle\!\langle \sigma, S, \mathit{expr} \rangle\!\rangle \Downarrow \langle\!\langle \sigma', S', \theta \rangle\!\rangle$, then $\gamma \Vdash \sigma \blacktriangleleft^{C}_{\mathcal{A}} \gamma' \Vdash \sigma'$ and $S \blacktriangleleft^{C}_{\mathcal{A}} S'$. However, this definition implicitly assumes $\theta = \mathit{no\text{-}br}$! For example, if $\theta = j + 1$ then a branch executed in expr and the stack σ' is not well-typed with respect to γ' anymore. We take this dependency of the type of σ' on θ with the following definition.

Definition 8 (θ-Variant Typing Contexts).

$$\Delta(C, \gamma, \theta) \triangleq \begin{cases} \gamma & \text{if } \theta = \mathit{no\text{-}br} \\ \mathit{merge}(C, \gamma, j) & \text{if } \theta = j \\ \langle C.\mathsf{return}, \gamma[0].\mathsf{snd} \rangle & \text{if } \theta = \mathit{return}, \end{cases}$$

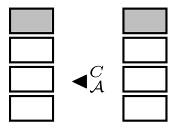
where $\mathit{merge}(C, \gamma, j) \triangleq \langle C.\mathsf{labels}[j] :: \gamma[j{+}1].\mathsf{fst}, \gamma[0].\mathsf{snd} \sqcup \gamma[j{+}1].\mathsf{snd} \rangle :: \gamma[j{+}2{:}]$.

Finally, we introduce an order on θs to capture the fact that if we branch in a high context we know something about the pc-labels in the output γ. Specifically, we have $\mathit{no\text{-}br} < 0 < 1 < \ldots < \mathit{return}$. We also need to define a translation of θs to integers with infinity where $\mathsf{nat}(\mathit{no\text{-}br}) = -1$, $\mathsf{nat}(j) = j$, and $\mathsf{nat}(\mathit{return}) = \infty$.

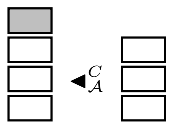
We are now ready to state our confinement lemma.

Lemma 1 (Confinement). *For any typing context C, store S_0, operand stack σ_0, stack-of-stacks γ_0, and expression expr, such that $C \vdash S_0$, $C \vdash \sigma_0$, and $\gamma_0 \Vdash \sigma_0$, if $\langle\!\langle \sigma_0, S_0, \mathit{expr} \rangle\!\rangle \Downarrow \langle\!\langle \sigma_1, S_1, \theta \rangle\!\rangle$, $\gamma_0, C \vdash \mathit{expr} \dashv \gamma_1$, and $\gamma_0[0].\mathsf{snd} \not\sqsubseteq \mathcal{A}$, then the following statements hold:*

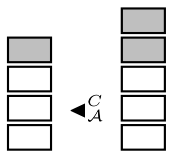
1. $\gamma_0 \Vdash \sigma_0 \blacktriangleleft^{C}_{\mathcal{A}} \Delta(C, \gamma_1, \theta) \Vdash \sigma_1$,
2. $S_0 \blacktriangleleft^{C}_{\mathcal{A}} S_1$, *and*
3. $\gamma_1[0 : \mathsf{nat}(\mathsf{pred}(\theta))].\mathsf{snd} \not\sqsubseteq \mathcal{A}$.



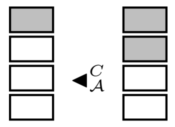
(a) Performing a step in the same block



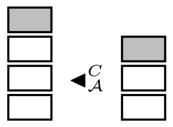
(b) Leaving normally the high context block



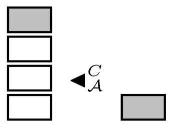
(c) Entering a block



(d) Conditional branching not taken



(e) Unconditional branching



(f) Return

Fig. 11: Pictorial representation of the confinement lemma. Each box represents an element $\langle st, pc\rangle$ of γ before (left) or after (right) the execution in the high context. White means $pc \sqsubseteq \mathcal{A}$, gray $pc \not\sqsubseteq \mathcal{A}$.

The confinement lemma as stated above and proven in the technical report [6], captures the intuition laid out previously. Furthermore, the different cases one needs to consider in the proof are illustrated in Figure 11.

Noninterference Next we turn our attention to stating and proving noninterference. We would like to state a classical theorem along the lines "if you start off with two $\mathcal{A}$-equivalent configurations and execute the same program in both, you end up with two $\mathcal{A}$-equivalent configurations." However, this is not a strong enough statement to induct over the evaluation of expressions in SecWasm because the two different executions may end up branching differently in a high context. For this reason we need a weaker notion of stack similarity than the strong equivalence given above.

Definition 9 (Weak Stack Similarity). *Stacks σ_0 and σ_1 with respective thetas θ_0 and θ_1 are weakly similar given γ and C (written $WS_{\gamma,C}(\langle\sigma_0,\theta_0\rangle,\langle\sigma_1,\theta_1\rangle)$) iff $\Delta(\gamma,C,\theta_0) \Vdash \sigma_0 \blacktriangleleft^C_\mathcal{A} \Delta(\gamma,C,\theta_1) \Vdash \sigma_1$ or $\Delta(\gamma,C,\theta_1) \Vdash \sigma_1 \blacktriangleleft^C_\mathcal{A} \Delta(\gamma,C,\theta_0)\gamma \Vdash \sigma_0$, and if $\theta_0 \neq \theta_1$ then $\gamma[0 : |\mathsf{pred}(max(\theta_0,\theta_1))|].\mathsf{snd} \not\sqsubseteq \mathcal{A}$.*

This is enough to let us state and prove a sufficiently strong noninterference statement:

Theorem 1 (Noninterference). *If*
1. $\gamma, C \vdash expr \dashv \gamma'$,
2. $C \vdash S_0$ *and* $C \vdash S_1$,
3. $C \vdash \sigma_0$ *and* $C \vdash \sigma_1$,
4. $\gamma \Vdash \sigma_0 \sim^C_\mathcal{A} \gamma \Vdash \sigma_1$,
5. $\ll\sigma_0, S_0, expr\gg \Downarrow \ll\sigma_0', S_0', \theta_0\gg$ *and* $\ll\sigma_1, S_1, expr\gg \Downarrow \ll\sigma_1', S_1', \theta_1\gg$, *and*
6. $S_0 \sim^C_\mathcal{A} S_1$,

then $S'_0 \sim^C_{\mathcal{A}} S'_1$ *and* $WS_{\gamma',C}(\langle\sigma'_0, \theta_0\rangle, \langle\sigma'_1, \theta_1\rangle)$.

Finally, we note this theorem gives us a corollary resembling a traditional noninterference theorem.

Corollary 1 (Termination Insensitive Noninterference). *If*

1. $\langle st, pc\rangle, C \vdash expr \dashv \langle C.\mathsf{return}, pc'\rangle$,
2. $C \vdash S_0$ *and* $C \vdash S_1$,
3. $C \vdash \sigma_0$ *and* $C \vdash \sigma_1$,
4. $\langle st, pc\rangle \Vdash \sigma_0 \sim^C_{\mathcal{A}} \langle st, pc\rangle \Vdash \sigma_1$,
5. $\ll\sigma_0, S_0, expr\gg \Downarrow \ll\sigma'_0, S'_0, \theta_0\gg$ *and* $\ll\sigma_1, S_1, expr\gg \Downarrow \ll\sigma'_1, S'_1, \theta_1\gg$, *and*
6. $S_0 \sim^C_{\mathcal{A}} S_1$,

then $S'_0 \sim^C_{\mathcal{A}} S'_1$ *and* $\langle C.\mathsf{return}, pc'\rangle \Vdash \sigma'_0 \sim^C_{\mathcal{A}} \langle C.\mathsf{return}, pc'\rangle \Vdash \sigma'_1$.

This corollary holds because if the program *expr* terminates without trapping, then it terminates with either $\theta = \textit{no-br}$ or $\theta = \textit{return}$ and both of these guarantee that the two output stacks are typed *with the same stack type.* When they do, $\blacktriangleleft^C_{\mathcal{A}}$ boils down to $\sim^C_{\mathcal{A}}$.

6 Discussion

Several points we have not addressed in the paper are worth discussing. These are implementation, overhead, usability, and declassification. Before addressing them below, we stress that they are extensions to our work and important avenues for future exploration and not mandatory for foundational IFC in Wasm.

Implementation and Overhead It is difficult to judge the overhead our framework would entail without having an actual implementation. We have argued for and justified the hybrid design of SecWasm as a trade-off between achieving permissiveness and expressiveness, and incurring some runtime overhead. While the semantics carry only few dynamic checks, the type system is heavily populated with additional IFC constraints which might slow-down the type-checking mechanism. However, as in prior work, the concern is not on the static overhead, but on the dynamic one. As we keep dynamic checks to a minimum, we are confident future benchmarks will not reveal considerate overheads.

Usability We expect the use of SecWasm to be straightforward. The developer would have to manually annotate the function types and the **load** and **store** operations with security labels, and then to verify if any detected illicit information flows are due to buggy implementations or imported malicious modules (such as the password meter module *PM*).

Declassification Certain situations require sensitive data to be released, an operation known as declassification [31]. When designing a declassification mechanism, one should aim to have it *robust*, meaning not allowing public data to influence what data to be declassified [32].

Sabelfeld and Sands presented four dimensions of declassification: *what* information is released, *who* is releasing information, *where* in the system information is released, and *when* information can be released [37]. To allow declassification in a static IFC system for Wasm, Watt *et al.* allowed functions marked as trusted to declassify data through a declassification primitive [49]. In order to extend SecWasm with a declassification construct, the formalization of the security properties enforced by the current system must be altered, as some information about the secret data could be learned by a public observer. In this sense, a password checker is different from a password meter because the latter leaks some information about the password. Although we leave it for future work, we believe our approach can be straightforwardly extended to handle the *what* dimension from Sabelfeld and Sands by guaranteeing that the system cannot leak more secrets than allowed by externally-specified escape hatches.

7 Related work

IFC for Low-Level Languages There has been much work on securing (subsets of) Java bytecode [25,20,11,7,5], or on enforcing security in TAL (Typed Assembly Language) [30,29,13,52,21] which models the RISC architecture, and even on JavaScript bytecode [10]. These approaches dealt with languages with unstructured control flow and heap memory, with TAL also employing registers. Due to lack of structured control flow at the low-level, prior work resorted to mimicking the block structure of the original high-level languages and computing dependence regions: linear continuations and continuation stacks [13], static code labels [29], control regions [5,25,10], type annotations [29,52]. Due to the structured control flow inherited from Wasm, in SecWasm the language's constructs proved sufficient for computing the dependence regions.

Most previous approaches dealt with Java bytecode or TAL, both languages without dynamic features. Thus, the preferred IFC enforcement was static, through security type systems [13,29,5,25,52]. More recently, a hybrid system was suggested for TAL-like languages [21], in an attempt to increase permissiveness over previous fully static approaches. Due to being a language heavily-charged with dynamic features, JavaScript bytecode was instrumented through a dynamic monitor, although prior static analysis is required for computing the control flow graphs and immediate post-dominators [10]. Although Wasm does not exhibit the same dynamism as JavaScript does, the nature of memory accesses requires a dynamic handling if a more expressive and permissive system is desired. Thus, SecWasm is designed to be mainly static and introduces dynamic checks in key places to increase permissiveness.

Cassel *et al.* present FlowNotation to find information flow violations in C programs [15], and De Francesco and Martini use abstract interpretation for instruction-level information-flow analysis [16]. Both have similar handling of the memory as SecWasm. With FlowNotation, each pointer (i.e., heap location) and its corresponding value are labeled with security policies which are joined upon dereferencing the pointer, and De Francesco and Martini label each memory location with a label to represent the maximum security level of the data to be stored. However, since FlowNotation does not handle pointer arithmetic and the memory in the system by De Francesco and Martini is a map of variables to abstract values, neither of those solutions have an unstructured memory as in SecWasm with partial re-writes of data (such as Example 3, where part of the 32-bit integer value starting at position 0 is overwritten).

Hybrid IFC While hybrid analyses were not so popular amongst low-level languages, they have been employed for high-level languages [46,26,35,8,23]. Our hybrid mechanism draws on the basic principles laid out in prior work, such as establishing what paths are reachable by dynamic analysis and inferring what dependencies arise from non-taken branches by static analysis [26,35]. A key contribution of SecWasm is extending these principles to deal with the challenges of an unstructured linear memory.

Wasm Security Lehmann *et al.* [27] prove vulnerabilities with well-known mitigations in the original high-level code propagate down to Wasm code. As a vulnerable program in C/C++ compiled to Wasm can translate the memory vulnerabilities, Disselkoen *et al.* introduce MS-Wasm, an extension to Wasm allowing developers to capture low-level C/C++ memory semantics in Wasm at compile time [18]. Swivel is a compiler framework to harden Wasm against Spectre attacks [33]. These works, however, do not focus on information-flow control.

Different language-based security techniques for Wasm perform taint-tracking. Szanto *et al.* propose a Wasm virtual machine in JavaScript [43], TaintAssembly presents a taint-tracking engine for interpreted Wasm implemented in V8 [19], while Wasabi is an expressive framework for dynamically analyzing and taint-tracking in Wasm [28]. Lastly, Stiévenart and De Roover [41] use taint-tracking to create function summaries, i.e., descriptions of where information from the function parameters and global variables can flow to when a function is invoked. Compared to these techniques, SecWasm not only tracks explicit and implicit flows, but also memory accesses.

Vivienne is an open-source tool that performs symbolic analysis and constraint solving for analyzing constant-time properties in Wasm programs [45]. Watt *et al.* introduce CT-Wasm [49], a type-driven extension to Wasm for constant-time cryptographic applications. To achieve constant-time, CT-Wasm disallows secret-dependent control instructions, being thus more restrictive than SecWasm. Furthermore, CT-Wasm introduces a separate memory for storing secret data, while in SecWasm we annotate individual memory cells with security labels, an approach that scales to general lattices.

Gradual Typing Gradual typing allows programmers to control the combination of dynamic and static approaches *at the programming level* [39]. Swamy *et al.* [42] presented TS* that adds a static static type system over JavaScript and Rastogi *et al.* [34] presented Safe TypeScript to catch any dynamic type errors while not altering the semantics of type-safe TypeScript code.

Gradual typing has also been used for IFC. Disney and Flanagan described an IFC type system for λ-calculus that defers cast checks that cannot be determined statically to the runtime [17]. In HLIO, Buiras *et al.* used gradual typing to allow programmers to defer some IFC checks to runtime in Haskell [14]. Bichhawat *et al.* investigated the tension between noninterference and gradual guarantees and defined a simple imperative languages that provides both noninterference and gradual guarantees [9].

Although there are high-level connections with gradual typing, there are also important differences. Indeed, gradual typing gives the developer the control of when to use static and when to use dynamic types. In our approach, the split is taken care of by the enforcement mechanism.

8 Conclusions

This paper presented SecWasm, the first general-purpose information-flow enforcement mechanism for Wasm. The synergy of static and dynamic IFC enforcement in SecWasm is the result of a thorough design analysis that leverages the already existing Wasm type system, while also ensuring permissiveness for Wasm's dynamic features. SecWasm overcomes the challenges imposed by the combination of uncommon characteristics for machine languages of structured control flow and linear memory in an elegant way. Finally, SecWasm provably enforces termination-insensitive noninterference.

In line with other foundational work on hybrid IFC (e.g., [26,35,8,23]), we leave implementation and experiments with performance overhead as an important track for future work.

Acknowledgements This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), Meta, the European Research Council (ERC) under the Horizon 2020 research (grant 771527-BROWSEC), and by the Vienna Business Agency through the project Vienna Cybersecurity and Privacy Research Center (VISP).

References

1. Ethereum WebAssembly (ewasm). `https://ewasm.readthedocs.io/en/mkdocs/`.
2. WebAssembly Security. `https://webassembly.org/docs/security/`.
3. M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. *TISSEC*, 2009.
4. A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *ESORICS*, 2008.
5. G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight Non-Interference Java Bytecode Verifier. *Math. Struct. Comput. Sci.*, 2013.
6. I. Bastys, M. Algehed, A. Sjösten, and A. Sabelfeld. SecWasm: Information Flow Control in WebAssembly—Full version. `https://www.cse.chalmers.se/research/group/security/secwasm/`, 2022.
7. C. Bernardeschi and N. De Francesco. Combining Abstract Interpretation and Model Checking for Analysing Security Properties of Java Bytecode. In *VMCAI*, 2002.

8. F. Besson, N. Bielova, and T. P. Jensen. Hybrid Information Flow Monitoring against Web Tracking. In *CSF*, 2013.
9. A. Bichhawat, M. McCall, and L. Jia. Gradual Security Types and Gradual Guarantees. In *CSF*, 2021.
10. A. Bichhawat, V. Rajani, D. Garg, and C. Hammer. Information Flow Control in WebKit's JavaScript Bytecode. In *POST*, 2014.
11. P. Bieber, J. Cazin, P. Girard, J. Lanet, V. Wiels, and G. Zanon. Checking Secure Interactions of Smart Card Applets: Extended Version. *J. Comput. Secur.*, 2002.
12. A. Birgisson, A. Russo, and A. Sabelfeld. Unifying Facets of Information Integrity. In *ICISS*, 2010.
13. E. Bonelli, A. Compagnoni, and R. Medel. SIFTAL: A Typed Assembly Language for Secure Information Flow Analysis. Technical report, 2004.
14. P. Buiras, D. Vytiniotis, and A. Russo. HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell. In *ICFP*, 2015.
15. D. Cassel, Y. Huang, and L. Jia. FlowNotation: Uncovering Information Flow Policy Violations in C Programs. *CoRR*, abs/1907.01727, 2019.
16. N. De Francesco and L. Martini. Instruction-level security analysis for information flow in stack-based assembly languages. *Inf. Comput.*, 2007.
17. T. Disney and C. Flanagan. Gradual Information Flow Typing. In *STOP*, 2011.
18. C. Disselkoen, J. Renner, C. Watt, T. Garfinkel, A. Levy, and D. Stefan. Position Paper: Progressive Memory Safety for WebAssembly. In *HASP@ISCA*, 2019.
19. W. Fu, R. Lin, and D. Inge. TaintAssembly: Taint-Based Information Flow Control Tracking for WebAssembly. *CoRR*, abs/1802.01050, 2018.
20. S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In *VMCAI*, 2005.
21. E. Geraldo, J. F. Santos, and J. C. Seco. Hybrid Information Flow Control for Low-Level Code. In *SEFM*, 2021.
22. A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. F. Bastien. Bringing the Web up to Speed with WebAssembly. In *PLDI*, 2017.
23. D. Hedin, L. Bello, and A. Sabelfeld. Value-Sensitive Hybrid Information Flow Control for a JavaScript-Like Language. In *CSF*, 2015.
24. K. Hoffman. WebAssembly in the Cloud. `https://medium.com/@KevinHoffman/webassembly-in-the-cloud-2f637f72d9a9`.
25. N. Kobayashi and K. Shirane. Type-Based Information Analysis for Low-Level Languages. In *APLAS*, 2002.
26. G. Le Guernic. Automaton-based Confidentiality Monitoring of Concurrent Programs. In *CSF*, 2007.
27. D. Lehmann, J. Kinder, and M. Pradel. Everything Old is New Again: Binary Security of WebAssembly. In *USENIX Security*, 2020.
28. D. Lehmann and M. Pradel. Wasabi: A Framework for Dynamically Analyzing WebAssembly. In *ASPLOS*, 2019.
29. R. Medel, A. B. Compagnoni, and E. Bonelli. A Typed Assembly Language for Non-interference. In *ICTCS*, 2005.
30. J. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Trans. Progr. Lang. Sys.*, 1999.
31. A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *SOSP*, 1997.
32. A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing Robust Declassification and Qualified Robustness. *J. Comput. Secur.*, 2006.

33. S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. M. Tullsen, and D. Stefan. Swivel: Hardening WebAssembly against Spectre. In *USENIX Security*, 2021.
34. A. Rastogi, N. Swamy, C. Fournet, G. M. Bierman, and P. Vekris. Safe & Efficient Gradual Typing for TypeScript. In *POPL*, 2015.
35. A. Russo and A. Sabelfeld. Dynamic vs. Static Flow-Sensitive Security Analysis. In *CSF*, 2010.
36. A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *JSAC*, 2003.
37. A. Sabelfeld and D. Sands. Declassification: Dimensions and Principles. *J. Comput. Secur.*, 2009.
38. J. F. Santos, T. P. Jensen, T. Rezk, and A. Schmitt. Hybrid Typing of Secure Information Flow in a JavaScript-Like Language. In P. Ganty and M. Loreti, editors, *Trustworthy Global Computing - 10th International Symposium, TGC 2015, Madrid, Spain, August 31 - September 1, 2015 Revised Selected Papers*, volume 9533 of *Lecture Notes in Computer Science*, pages 63–78. Springer, 2015.
39. J. Siek and W. Taha. Gradual Typing for Functional Languages. *Scheme and Functional Programming Workshop (SFP)*, 2006.
40. R. G. Singh and C. Scholliers. WARDuino: a Dynamic WebAssembly Virtual Machine for Programming Microcontrollers. In *MPLR*, 2019.
41. Q. Stiévenart and C. De Roover. Compositional Information Flow Analysis for WebAssembly Programs. In *SCAM*, 2020.
42. N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P. Strub, and G. M. Bierman. Gradual Typing Embedded Securely in JavaScript. In *POPL*, 2014.
43. A. Szanto, T. Tamm, and A. Pagnoni. Taint Tracking for WebAssembly. *CoRR*, abs/1807.08349, 2018.
44. J. Szefer. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *J. of Hardware and Sys. Sec.*, 2019.
45. R. Tsoupidi, M. Balliu, and B. Baudry. Vivienne: Relational Verification of Cryptographic Implementations in WebAssembly. In *IEEE Secure Development Conference, SecDev 2021, Atlanta, GA, USA, October 18-20, 2021*, pages 94–102. IEEE, 2021.
46. P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *NDSS*, 2007.
47. W3C. WebAssembly Core Specification. `https://www.w3.org/TR/wasm-core-1/`.
48. L. Wagner. WebAssembly Consensus and End of Browser Preview. `https://lists.w3.org/Archives/Public/public-webassembly/2017Feb/0002.html`.
49. C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan. CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem. *POPL*, 2019.
50. WebAssembly Community Group. WebAssembly Specification, current version. `https://webassembly.github.io/spec/core/`.
51. E. Wen and G. Weber. Wasmachine: Bring IoT up to Speed with A WebAssembly OS. In *PerCom Workshops*, 2020.
52. D. Yu and N. Islam. A Typed Assembly Language for Confidentiality. In *ESOP*, 2006.
53. S. A. Zdancewic. *Programming languages for information security.* PhD Thesis, Cornell University, 2002.