Compositional Information Flow Analysis for WebAssembly Programs

Quentin Stiévenart, Coen De Roover
Software Languages Lab, Vrije Universiteit Brussel, Belgium
{quentin.stievenart,coen.de.roover}@vub.be

***Abstract*—WebAssembly is a new W3C standard, providing a portable target for compilation for various languages. All major browsers can run WebAssembly programs, and its use extends beyond the web: there is interest in compiling cross-platform desktop applications, server applications, IoT and embedded applications to WebAssembly because of the performance and security guarantees it aims to provide. Indeed, WebAssembly has been carefully designed with security in mind. In particular, WebAssembly applications are sandboxed from their host environment. However, recent works have brought to light several limitations that expose WebAssembly to traditional attack vectors. Visitors of websites using WebAssembly have been exposed to malicious code as a result.**

In this paper, we propose an automated static program analysis to address these security concerns. Our analysis is focused on information flow and is compositional. For every WebAssembly function, it first computes a summary that describes in a sound manner where the information from its parameters and the global program state can flow to. These summaries can then be applied during the subsequent analysis of function calls. Through a classical fixed-point formulation, one obtains an approximation of the information flow in the WebAssembly program. This results in the first compositional static analysis for WebAssembly. On a set of 34 benchmark programs spanning 196kLOC of WebAssembly, we compute at least 64% of the function summaries precisely in less than a minute in total.

***Index Terms*—Static program analysis, WebAssembly, security**

I. Introduction

WebAssembly [31] "*is a binary instruction format for a stack-based virtual machine*" [6], designed as a compilation target for high-level languages. The specification of its core has been a W3C standard since December 2019 [7]. WebAssembly was designed for the purpose of embedding binaries in web applications, thereby enabling intensive computations on the web. Current trends however show that the usage of WebAssembly goes beyond web applications. It is now used for cross-platform desktop applications, thanks to its ability to easily incorporate functions that are provided by the runtime. In particular, the WebAssembly System Interface (WASI) [12] is an API that focuses on providing such functions to WebAssembly applications, in order to deal with files and networking. As long as the proper functions are provided, WebAssembly can also be used for IoT backends [32] and for embedded systems [55]. Finally various runtimes are being developed for WebAssembly [1], [3]–[5], [51], [52].

WebAssembly is quickly gaining in popularity: a 2019 study [44] demonstrated that 1 in 600 websites among the top 1 million Alexa websites rely on WebAssembly. However, the same study revealed an alarming finding: in 2019, the most common application of WebAssembly is to perform *cryptojacking*, i.e., relying on the visitor's computing resources to mine cryptocurrencies without authorisation. Moreover, despite being designed with security in mind, WebAssembly applications are still vulnerable to several traditional security attacks, on multiple execution platforms [37].

Consequently, there needs to be proper tool support for preventing and identifying malicious usage of WebAssembly. There has been some early work on improving the safety and security of WebAssembly, e.g., through improved memory safety [22], code protection mechanisms [59], and sandboxing [28]. Also, dynamic analyses have been proposed for detecting cryptojacking [16], [67] or for performing taint tracking [25], [60]. However, not a single *static* analysis for WebAssembly has been proposed so far.

In this paper, we aim to fill this void by providing the foundations for a method of static analysis tailored to WebAssembly. A number of challenging problems need to be overcome. First, traditional whole-program analyses may not be used, as WebAssembly binaries are often libraries of which the usage is not known statically. Second, static analysis of binaries is challenging due to their low-level nature [41], [54]. Finally, for the sake of applicability, the analysis of WebAssembly binaries has to be fast and automated so it can be incorporated by runtimes to identify potential security problems before running the application under analysis.

To address these challenges, we turn to a *compositional*, *summary-based* analysis method [20]. In such an analysis, code segments are analysed in isolation from each other and the analysis computes a *summary* of their individual behaviour. Our analysis aligns code segments with WebAssembly functions, and computes summaries that describe how information flows across the execution of functions. We deliberately do not model flow through the linear memory of WebAssembly nor traps as part of the summaries, and we focus on explicit information flows. We observe that, in practice, this does not result in soundness issues on 34 benchmark programs.

This paper makes the following contributions:

- We present the first compositional static analysis for WebAssembly, in the form of a static information flow analysis. This analysis is compositional in nature, and derives function summaries. These summaries approximate the information flow within a function, and are used to conduct an inter-procedural analysis in a compositional manner.

- We demonstrate how a static analysis for WebAssembly can benefit from the design of WebAssembly to statically over-approximate indirect function calls.
- We implemented our analysis within a framework that we will open source. We evaluated our analysis on 34 WebAssembly programs, totalling 196kLOC, which were analysed in less than 2 minutes, with a precision of 64%.

II. Motivation

In this section, we motivate the desiderata for any static analysis of WebAssembly and highlight the peculiarities of WebAssembly that distinguish it from other assembly languages in terms of static analysis support.

A. Analysis with Security Applications

Safety has been a focal point in the design of WebAssembly, as mentioned in the documentation: "*the design of WebAssembly promotes safe programs by eliminating dangerous features from its execution semantics*" [8]. However, suitability as an efficient compilation target for languages such as C and C++ has received equal consideration. A linear memory model [22] is the key feature to which WebAssembly owes this suitability, meaning that memory is represented as an array of bytes that can be read from and written to without limitations. To ensure safety, WebAssembly programs —and their linear memory— are isolated from the host execution environment: an incorrect manipulation of the linear memory cannot result in arbitrary code being executed on the host execution environment. However, they are still prone to the usual vulnerabilities such as buffer overflows within their own execution. Similarly, although the isolation of the WebAssembly executable limits the possibilities for remote code execution, it is feasible [40] nonetheless through a combination of indirect function calls, buffer overflow, and code executing functions such as Emscripten's `emscripten_run_script` which can run arbitrary JavaScript code on the browser. Lehmann et al. [37] have even demonstrated that WebAssembly binaries are vulnerable to various attacks, and that this is the case on several execution platforms.

B. Static Analysis of Binaries

To address the increasing security concerns, WebAssembly refinements such as improved memory safety [22] and two-way sandboxing [28] as well as code protection mechanisms [59] have been proposed. The first dynamic analyses for vulnerability detection have also been proposed, against cryptojacking [67] and unwanted information flow [25], [60]. To this date, however, no automated static analysis has been proposed to protect the browser or WebAssembly runtime from executing malicious instructions.

WebAssembly is a binary instruction language by definition. For the browser or runtime executing WebAssembly instructions, the original source code of the compiled program is no longer available. On the one hand, this is unfortunate as source code can often be analysed with higher precision as it reflects developer intentions more directly. On the other hand, analysing a smaller binary language comes with the advantage that its semantics is often well-defined. The core language specification of WebAssembly is not only a W3C standard, but it has also been mechanically formalised [68]. Notably, the WebAssembly specification does not contain any undefined behaviour.

Static data flow analysis of x86 binaries is notoriously difficult, mostly due to the difficulty of constructing the required Control Flow Graph (CFG) in the presence of arbitrary jump instructions. Several data flow analyses [23], [35], [45], [54] dedicate a separate and complex analysis to this task, or have to refine a pre-computed CFG during their own analysis [14]. The design of WebAssembly alleviates this challenge. Branching instructions unambiguously identify the targets of their jumps, either as a single target (with the `br` instruction), two targets (`br_if`), or a static list of targets (`br_table`). Moreover, function calls —another form of branching— come in two forms: the `call` i instruction calls the function identified by index i (statically defined), while the `call_indirect` t instruction calls the function of type t, of which the index is at a location in a statically defined table, given by the value on the top of the stack. Hence, the `call_indirect` instruction is the only unknown in the control flow of WebAssembly. By relying on type information, the targets of an indirect call can be approximated, enabling the construction of an approximate call graph before analysis.

C. Compositional Analysis

A WebAssembly module is defined as a set of functions, some of which are *exported* and made available to the host execution environment, some of which are *imported* and are made available by the host execution environment to be called from within the WebAssembly module. This bi-directional interface enables replacing JavaScript libraries with more efficient binaries, which is a prevalent practice as Musch et al. found that 39% of the top 1 million Alexa websites that rely on WebAssembly do so in the form of binary libraries [44].

Hence, any whole-program analysis would require knowing how a WebAssembly module is used by the client code, and needs to support analysing both client-side JavaScript code and WebAssembly code. This would be a tremendous engineering effort, and would potentially result in slow analyses, as whole-program analyses are known to face scalability issues [30], [72]. Instead, we argue that some form of *compositional* analysis is required. In a compositional analysis, the approximation of the semantics of an entire program is obtained by composing the approximations of the semantics of the program's parts [20]. As a result, a compositional analysis can not only analyse whole programs, but can also analyse portions of programs from any entry point, which fits the need for analysing libraries where an entry point is a function call. Compositional analyses, and similar forms of modular analyses, have been shown to scale well [17], [24], [27], [29], [33], [43], [58], [71].

A summary-based compositional analysis approximates the semantics of a program part through a *summary*. In our case, the program parts that are analysed in isolation from each other are the functions of the WebAssembly program under analysis. The information maintained in the summary for a function needs to be sufficiently precise to support the purpose of the analysis. To enable determining unwanted information flow, a function summary should at least provide an approximation of what it returns in terms of its parameters and how it modifies the global variables (i.e., registers). It may also include information on how the linear memory is updated. As an example, consider the following excerpt of a WebAssembly module with one global variable. This is the definition of a function that takes one argument and returns a value. It first pushes the constant 1 on the value stack (line 3), then pushes the value of local variable 0 (line 4), which is the first parameter to the function. Finally, both values are added with the binary operation add, which reads both values from the stack and pushes the result on the stack (line 5). The top value of the stack after the last instruction is the return value of the function. Hence, this function returns the value of its parameter incremented by one, and does not modify the global variable.

```
1 (type (func i32 -> i32))
2 (func (type 1) (local)
3   i32.const 1 ;; stack is: [1]
4   local.get 0 ;; stack is: [arg0, 1]
5   i32.add)    ;; stack is: [arg0+1]
```

For this function, our summary-based compositional analysis produces the summary $(\{\mathbf{parameter}(0)\}, \{\mathbf{global}(0)\})$. This indicates that the return value (first element of the tuple) contains information from the first parameter of the function, and that the global variable after the execution of the function only contains information from the global variable before the execution. This summary holds regardless of the values of its parameters, the values of the global variables, and the content of the linear memory.

III. MiniWasm: A Minimal Version of WebAssembly

For presentation purposes, we introduce MiniWasm as a subset of WebAssembly that contains all defining features that should be supported by a static analysis. Our implementation is *not* limited to the MiniWasm subset, and supports the full WebAssembly language except for traps. We refer to the WebAssembly specification [7] and its mechanisation [68] for a formal treatment of the full language. MiniWasm retains the following defining features of WebAssembly:

- 32-bit integers.
- Instructions for structured control flow: blocks (block), loops (loop), and structured conditional jumps (br_if).
- Instructions to manipulate the WebAssembly runtime: the local variables (local.get, local.set), the global variables (global.get, global.set), and the linear memory (load, store).
- Instructions for direct and indirect function calls (call, call_indirect).
- Instructions for unary and binary operations (*unop* and *binop*).
- Imported functions that are left undefined.
- Function pointer indirection through a *table*.

A MiniWasm *module* is composed of a list of function types, a set of functions, and a table that identifies function pointers. Without loss of generality, MiniWasm does not contain the following features of the WebAssembly specification:

- Other datatypes: WebAssembly supports 64-bit integers, 32-bit floats, 64-bit floats, as well as various operations on these types.
- A WebAssembly module may declare the initial contents of the memory as part of its definition. The contents of the memory can then be modified, either during the execution of a WebAssembly function, or by the host environment. This is not included in MiniWasm, because we do not know how the memory may be changed by the host environment, and the results of a compositional analysis need to be valid for any change to that memory.
- MiniWasm does not feature exceptional control flow (traps) nor the return instruction.
- In WebAssembly, functions may have names: these are easier to read than indices when referring to a function in call instructions. Moreover, WebAssembly modules may export functions, giving them a name, such that they can be called from the host environment. MiniWasm does not contain any name.
- MiniWasm only features conditional branching (br_if), as other WebAssembly branching instructions (br, br_table) can be rewritten as conditional branching.
- In contrast to the actual WebAssembly specification, the linear memory of MiniWasm is a map from 32-bit integer values to 32-bit integer values, and the store and load operations do not have an extra parameter to specify alignments, signedness, and pack size.

A. Syntax of MiniWasm

The syntax of MiniWasm is defined below.

$$\begin{array}{rcl}
\mathit{module} & ::= & \texttt{(module } \mathit{type}^* \ \mathit{func}^* \ \mathit{table}\texttt{)} \\
\mathit{type} & ::= & \texttt{(type (func } \mathit{ft}\texttt{))} \\
\mathit{bt}, \mathit{ft} & ::= & t^* \rightarrow t^* \\
t & ::= & \texttt{i32} \\
\mathit{func} & ::= & \texttt{(func (type } \mathit{tidx}\texttt{))} \\
 & | & \texttt{(func (type } \mathit{tidx}\texttt{) (local } t^*\texttt{) } \mathit{instr}^*\texttt{)} \\
\mathit{table} & ::= & \texttt{(table } n^*\texttt{)} \\
\mathit{instr} & ::= & \mathit{data} \mid \mathit{control} \\
\mathit{data} & ::= & \texttt{drop} \mid t\texttt{.const } n \mid t.\mathit{binop} \mid t.\mathit{unop} \\
 & | & \texttt{local.get } n \mid \texttt{local.set } n \\
 & | & \texttt{global.get } n \mid \texttt{global.set } n \\
 & | & \texttt{load} \mid \texttt{store} \\
\mathit{control} & ::= & \texttt{block } \mathit{bt} \ \mathit{instr}^* \texttt{ end} \mid \texttt{loop } \mathit{bt} \ \mathit{instr}^* \texttt{ end} \\
 & | & \texttt{call } \mathit{ft} \ n \mid \texttt{call\_indirect } \mathit{ft} \mid \texttt{br\_if } l \\
n, l, \mathit{tidx} & ::= & \text{a number}
\end{array}$$

A MiniWasm module (definition *module* in the syntax) contains:

- A sequence of function type declarations (*type**).
- A sequence of function declarations (*func**).
- A table (*table*) that is used to identify targets of indirect function calls: the table is a sequence of function indices that may be called upon a `call_indirect` instruction.

A function can either be an *imported* function, defined by the host environment, or a function defined in the module. A function has a type, which is declared by providing its index *tidx* in the sequence of type declarations. A defined function may have local variables, and consists of a sequence of instructions. Local variables are not named but rather indexed, and are composed of first the function's parameters, followed by the declared locals: a function with one parameter and two local variables can access local variable 0 for accessing its parameter, and local variables 1 and 2 for accessing the declared local variables. Instructions can either be *data instructions*, which are instructions that manipulate the stack (`drop`, *const*), locals (`local.get` and `local.set`) or globals (`global.get` and `global.set`). Or, they can be *control instructions*, which influence the program's control flow. We leave *binop* and *unop* unspecified as our analysis will not distinguish between operators. Blocks (`block`) and functions are annotated with their types (respectively, *bt* and *ft*). We denote by bt_{in} (resp. bt_{out}) the input arity (resp. output arity) of a block type, and similarly for a function type. Blocks act as delimiters inside functions, for identifying jump targets. Below we illustrate MiniWasm through a few examples.

B. MiniWasm Examples

The following example demonstrates the use of structured jump instructions. The `br_if` instruction is used to jump out of a number of enclosing blocks, or to jump back the beginning of a loop. Here, `br_if 0` at line 9 will break out of the current block (ranging from line 4 to line 12) if the top value on the stack is non-zero. In general, `br_if n` breaks out of the `n`th parent block, or to the beginning of a loop as we will see in the next example.

```
1  (type (func i32 i32 -> i32))
2  (func (type 1)  (local)
3    ;; this block leaves one value on the stack
4    block -> i32
5      local.get 0 ;; stack is [arg0]
6      local.get 1 ;; stack is [arg1, arg0]
7      ;; breaks out of the block
8      ;; if arg1 is non-zero
9      br_if 0
10     drop ;; stack is []
11     local.get 1 ;;  stack is [arg1]
12   end)
13 ;; final stack is [arg1] if arg1 = 0
14 ;; otherwise it is [arg0]
```

The following example demonstrates the use of the loop construct and of local variables. The function takes one argument and produces one return value. To do so, it relies on two extra local variables (declared on line 3). The local variables are manipulated with instructions `local.get` and `local.set`. The local variables of a function are in fact a sequence formed by the parameters of the function followed by the declared local variables, which are initialised to 0 upon function entry. The behaviour of a break instruction in a `loop` block is to jump to the beginning of the loop. In case the end of the loop is reached without breaking, the loop ends its execution. This means that in the following code, the loop iterates while `local1-1` is non-zero.

```
1  (type (func i32 -> i32))
2  (func (type 1)
3    (local i32 i32)
4    local.get 0 ;; stack is [arg0]
5    local.set 1 ;; stack is []
6    loop
7      local.get 2 ;; stack is [local2]
8      i32.const 1 ;; stack is [1, local2]
9      i32.add     ;; stack is [local2+1]
10     local.set 2 ;; stack is []
11     local.get 1 ;; stack is [local1]
12     i32.const 1 ;; stack is [1, local1]
13     i32.sub     ;; stack is [local1-1]
14     local.set 1 ;; local1 is set to local1-1
15     local.get 1 ;; stack is [local1]
16     br_if 1
17   end
18   local.get 2)
```

The final example illustrates direct and indirect function calls. A direct function call is performed by placing the arguments on the stack (lines 4 and 5), followed by issuing the `call` instruction (line 8). The function called will be the function that has the corresponding index: in this case, the function with index 0 is called. Function indices are assigned to each function by following the order in which they are defined in the module: function 0 is the first function declared in the module. The return value of the called function will be on the top of the stack after the function call. Finally, the control flow returns to the caller once the called function has finished its execution. For indirect function calls, arguments are also placed on the stack (lines 10 and 11). Then, an index is placed on the stack (line 12) before performing the `call_indirect` instruction (line 13). This index will be looked up in the table (declared on line 14), and the corresponding function will be called. In this case, index 0 is looked up in the table: it points to function 0, which is therefore called. Note that the `call_indirect` instruction provides the type of the function to be called.

```
1  (type (func i32 i32 -> i32))
2  (func (type 1) (local) ...) ;; function 0
3  (func (type 1) (local)
4    i32.const 0
5    i32.const 1
6    ;; calls fun. 0 with [0,1] as arguments
7    ;; results in 0+1
8    call 0
9    drop ;; stack is now empty
10   i32.const 10
11   i32.const 15
12   i32.const 0
13   call_indirect i32 i32 -> i32)
14 (table
15   0 ;; "pointer" to function 0
16   1) ;; "pointer" to function 1
```

C. Semantics of MiniWasm

This formalisation is inspired by the implementation of the WebAssembly specification [9], which itself supports the full WebAssembly language. A *state* of the execution of a MiniWasm module consists of a stack of values (denoted V), a stack of *administrative instructions*, called the *administrative stack* (A), local variables (L), global variables (G), and the linear memory (M). Accessing the value in linear memory M at address v is denoted $M[v]$, and updating the value at address v_1 with value v_2 is denoted $M[v_1 \mapsto v_2]$. Administrative instructions are either a plain MiniWasm instruction (**plain**), or synthetic indications that a function is being invoked (**invoke**), that a function call is being executed (**frame**), that a break has happened (**br**), or labels that identify blocks and loops (**label**). The linear memory is represented as a mapping from values to values. Functions have a type (*ft*), a number of local variables (n), and a body which is a sequence of instructions.

$$\begin{aligned}
\mathit{state} \in \mathit{State} &= \mathit{VStack} \times \mathit{AStack} \times \mathit{Locals} \times \mathit{Globals} \times \mathit{Mem} \\
L \in \mathit{Locals} &= \mathit{Value}^* \\
G \in \mathit{Globals} &= \mathit{Value}^* \\
v \in \mathit{Value} &= \mathbb{I}^{32}, \text{ the set of 32-bit integers} \\
V \in \mathit{VStack} &= \mathit{Value}^* \\
A \in \mathit{AStack} &= \mathit{AInstr}^* \\
M \in \mathit{Mem} &= \mathit{Value} \rightarrow \mathit{Value} \\
\mathit{ainstr} \in \mathit{AInstr} &::= \textbf{plain}(\mathit{instr}) \\
&\quad\mid \textbf{invoke}(\mathit{fun}, \mathit{fidx}) \mid \textbf{frame}(n, F, V, A, \mathit{fidx}) \\
&\quad\mid \textbf{br}(n, V) \mid \textbf{label}(n, \mathit{instr}^*, V, A) \\
\mathit{fun} &::= (\mathit{ft}, n, \mathit{instr}^*) \\
\mathit{fidx}, n &\text{ are integers}
\end{aligned}$$

The execution of a MiniWasm module is performed by the transition function defined in Figure 1. We use the following notation for sequences: a^* and A are sequences of a, $a \cdot A$ prepends element a to the sequence A, $A \cdot A'$ concatenates two sequences, A_i is the ith element of a sequence, $\mathsf{take}(n, A)$ keeps only the first n elements from a sequence, $\mathsf{drop}(n, A)$ drops the first n elements from a sequence, and $\mathsf{rev}(A)$ is the reverse of sequence A. A sequence of 0, of length n, can be constructed by $\mathsf{zeros}(n)$.

Data Instructions. `drop` removes the top value from the value stack. `i32.const` pushes a constant on the stack. For unary and binary operations, we assume that the behaviour of `i32.`*unop* (resp. `i32.`*binop*) is described by function unary (resp. binary). Accessing and modifying locals and globals is performed by accessing or updating the corresponding sequence of values (L or G). Loading from and storing to the linear memory is a matter of accessing or updating the memory mapping M.

Blocks, Loops, and Breaks. The semantics for `block` and `loop` instructions is to place a label on the administrative stack, containing the body instructions to execute and the number of values to keep from the stack when breaking from the label. Breaking means jumping either *out* of the nth parent block (hence the number of values kept on the stack is the out arity of the block), or back to the beginning of the nth parent loop (hence the number of values kept on the stack is the in arity of the loop). The value stack within the label only contains the required number of values to execute the corresponding body. The notation $\textbf{plain}(\mathit{instr})^*$ in the transition rule means that all instructions of the sequence instr^* are wrapped in a plain instruction. In case the top value of the stack is not 0, a `br_if` instruction results in a **br** administrative instruction being pushed, with the same level as requested by the `br` instruction. Otherwise, `br_if` is a no-op.

Function Calls and Frames. A `call` instruction extracts the body of the function called with the helper function $\mathsf{function}$ (which we assume has access to the function definitions of the module), and marks the function call through the **invoke** administrative instruction. In contrast, a `call_indirect` instruction looks up the module table to extract the function in the table of the module at the index given by the value at the top of the stack. We assume function $\mathsf{table\text{-}lookup}$ has access to the module table and returns the corresponding function index. An **invoke** administrative instruction creates a new frame. The locals of the frame are constructed by first extracting the arguments to the function from the stack, to which are appended the declared locals of the function, which are initialised to 0. The body of the function is itself wrapped in a label. In case all instructions in a frame have been executed, the frame has finished its execution and it is popped from the administrative stack. Otherwise, we continue stepping within the frame.

Labels. When all instructions within a label have been executed, the label can be removed. If a **br** administrative instruction has to be executed within a label, and the break level is 0, the current label is removed, the value stack is updated, and the instructions that have to be executed are placed on the administrative stack. For other break levels, the current label is removed and the level of the break is decreased. For any other instruction, an execution step is performed on the state inside the label.

IV. Instrumentation for Information Flow

We now instrument the semantics of MiniWasm to characterise what has to be approximated by the static analysis presented in this paper. We are interested in the flow of information from function parameters and global variables to return values of functions, and to the global variables after a function execution. To this end, the instrumentation annotates values in the state space with taint information:

$$\mathit{Value}^T = \mathbb{I}^{32} \times \mathit{TaintMap}$$

Taint information is a mapping from function indices to sets of taint sources: either parameters, or global variables. The bottom taint map $\bot$ assigns no taint to all function indices.

$$\begin{aligned}
\mathit{TaintMap} &= (\mathbb{N} \rightarrow \mathcal{P}(\mathit{Taint})) \\
\mathit{Taint} &::= \textbf{parameter}(n) \mid \textbf{global}(n) \\
\bot &= \lambda n.\varnothing
\end{aligned}$$

For example, the value $(1, [1 \mapsto \{\textbf{parameter}(0)\}, 2 \mapsto \{\textbf{parameter}(0), \textbf{parameter}(1)\}]$ means that the 32-bit integer

$$
\begin{array}{rrl}
\text{Data} & v \cdot V, \mathbf{plain}(\texttt{drop}) \cdot A, L, G, M & \rightarrow V, A, L, G, M \\
\text{instructions} & V, \mathbf{plain}(\texttt{i32.const}\ n) \cdot A, L, G, M & \rightarrow n \cdot V, A, L, G, M \\
 & v_2 \cdot v_1 \cdot V, \mathbf{plain}(\texttt{i32.}\mathit{binop}) \cdot A, L, G, M & \rightarrow \mathsf{binary}(\mathit{binop}, v_1, v_2) \cdot V, A, L, G, M \\
 & v \cdot V, \mathbf{plain}(\texttt{i32.}\mathit{unop}) \cdot A, L, G, M & \rightarrow \mathsf{unary}(\mathit{unop}, v) \cdot V, A, L, G, M \\
 & V, \mathbf{plain}(\texttt{local.get}\ l) \cdot A, L, G, M & \rightarrow L_l \cdot V, A, L, G, M \\
 & v \cdot V, \mathbf{plain}(\texttt{local.set}\ l) \cdot A, L, G, M & \rightarrow V, A, L_0 \cdots L_{l-1} \cdot v \cdot L_{l+1} \cdots L_n, G, M \\
 & V, \mathbf{plain}(\texttt{global.get}\ g) \cdot A, L, G, M & \rightarrow G_g \cdot V, A, L, G, M \\
 & v \cdot V, \mathbf{plain}(\texttt{global.set}\ g) \cdot A, L, G, M & \rightarrow V, A, L, G_0 \cdots G_{g-1} \cdot v \cdot G_{g+1} \cdots G_n, M \\
 & v \cdot V, \mathbf{plain}(\texttt{load}) \cdot A, L, G, M & \rightarrow M[v] \cdot V, A, L, G, M \\
 & v_2 \cdot v_1 \cdot V, \mathbf{plain}(\texttt{store}) \cdot A, L, G, M & \rightarrow V, A, L, G, M[v_1 \mapsto v_2] \\
 & & \\
\text{Blocks,} & V, \mathbf{plain}(\texttt{block}\ \mathit{bt}\ \mathit{instr}^*\ \texttt{end}) \cdot A, L, G, M & \rightarrow \mathsf{drop}(\mathit{bt}_{\mathit{in}}, V), \mathbf{label}(\mathit{bt}_{\mathit{out}}, \epsilon, \mathsf{take}(\mathit{bt}_{\mathit{in}}, V), \mathbf{plain}(\mathit{instr})^*) \cdot A, L, G, M \\
\text{loops,} & V, \mathbf{plain}(\texttt{loop}\ \mathit{bt}\ \mathit{instr}^*\ \texttt{end}) \cdot A, L, G, M & \rightarrow \mathsf{drop}(\mathit{bt}_{\mathit{in}}, V), \mathbf{label}(\mathit{bt}_{\mathit{in}}, \mathit{instr}^*, \mathsf{take}(\mathit{bt}_{\mathit{in}}, V), \mathbf{plain}(\mathit{instr})^*) \cdot A, L, G, M \\
\text{and breaks} & 0 \cdot V, \mathbf{plain}(\texttt{br_if}\ n) \cdot A, L, G, M & \rightarrow V, A, L, G, M \\
 & v \cdot V, \mathbf{plain}(\texttt{br_if}\ n) \cdot A, L, G, M & \rightarrow \epsilon, \mathbf{br}(n, V) \cdot A, L, G, M \text{ if } v \neq 0 \\
 & & \\
\text{Function} & V, \mathbf{plain}(\texttt{call}\ \mathit{ft}\ \mathit{fidx}) \cdot A, L, G, M & \rightarrow V, \mathbf{invoke}(\mathsf{function}(\mathit{fidx}), \mathit{fidx}) \cdot A, L, G, M \\
\text{calls} & v \cdot V, \mathbf{plain}(\texttt{call_indirect}\ \mathit{ft}) \cdot A, L, G, M & \rightarrow V, \mathbf{invoke}(\mathsf{function}(\mathit{fidx}), \mathit{fidx}) \cdot A, L, G, M \text{ where } \mathit{fidx} = \mathsf{table\text{-}lookup}(v) \\
\text{and frames} & V, \mathbf{invoke}((\mathit{ft}, n, \mathit{instr}^*), \mathit{fidx}) \cdot A, L, G, M & \rightarrow V, \mathbf{frame}(\mathit{ft}_{\mathit{out}}, L', \epsilon, \mathbf{label}(\mathit{ft}_{\mathit{out}}, \epsilon, \epsilon, \mathbf{plain}(\mathit{instr})^*), \mathit{fidx}) \cdot A, L, G, M \\
 & & \quad \text{where } L' = \mathsf{rev}(\mathsf{take}(\mathit{ft}_{\mathit{in}}, V)) \cdot \mathsf{zeros}(n) \\
 & V, \mathbf{frame}(n, _, V', \epsilon, \mathit{fidx}) \cdot A, L, G, M & \rightarrow V' \cdot V, A, L, G, M \\
 & V, \mathbf{frame}(n, L', V', A', \mathit{fidx}) \cdot A, L, G, M & \rightarrow V, \mathbf{frame}(n, L'', V'', A'', \mathit{fidx}) \cdot A, L, G', M' \text{ if } V', A', L', G, M \rightarrow V'', A'', L'', G', M' \\
 & & \\
\text{Labels} & V, \mathbf{label}(_, _, V', \epsilon) \cdot A, L, G, M & \rightarrow V' \cdot V, A, L, G, M \\
 & V, \mathbf{label}(n, \mathit{instr}^*, _, \mathbf{br}(0, V' \cdot _)) \cdot A, L, G, M & \rightarrow \mathsf{take}(n, V') \cdot V, \mathbf{plain}(\mathit{instr})^* \cdot A, L, G, M \\
 & V, \mathbf{label}(n, _, _, \mathbf{br}(n, V') \cdot _) \cdot A, L, G, M & \rightarrow V, \mathbf{br}(n-1, V') \cdot A, L, G, M \\
 & V, \mathbf{label}(n, \mathit{instr}^*, V', A') \cdot A, L, G, M & \rightarrow V, \mathbf{label}(n, \mathit{instr}^*, V'', A'') \cdot A, L', G', M' \text{ if } V', A', L, G, M \rightarrow V'', A'', L', G', M'
\end{array}
$$

Fig. 1. Semantics of MiniWasm. spaced according to the corresponding paragraphs in Section III-C.

value 1 has been computed in a way that is influenced by the first parameter of function 1 and by the first and second parameters of function 2: information flows from each of these parameters to this value. We only deal with explicit information flow in the present paper, which is in line with current research [49], [56], [57]. Taint maps are joined componentwise, e.g., $[1 \mapsto \{\mathbf{parameter}(0)\}] \sqcup [1 \mapsto \{\mathbf{parameter}(1)\} = [1 \mapsto \{\mathbf{parameter}(0), \mathbf{parameter}(1)\}]$.

A summary represents how information flows from parameters and global variables to the optional return value of a function (represented as a sequence of length 0 or 1, in the domain $\mathcal{P}(\mathit{Taint})^*$), and to the global variables after the execution of the function (in the domain $\mathcal{P}(\mathit{Taint})^*$). This is therefore represented as a tuple of which the first element is the taint of the optional return value, and the second element the taint of the global variables. The instrumented semantics computes a set of summaries S. In order to construct a summary for a function with index *fidx*, helper function summary extracts from the instrumented information of the optional return value (t_r) the taint from the point of view of the current function ($t_r[\mathit{fidx}]$), and similarly for the global variables after the function execution.

$$
\begin{aligned}
\mathit{Summary} &= \mathcal{P}(\mathit{Taint})^* \times \mathcal{P}(\mathit{Taint})^* \\
S \in \mathit{Summaries} &= \mathcal{P}(\mathit{Summary}) \\
\mathsf{summary}(\mathit{fidx}, (_, t_r)^*, (_, t_g)^*) &= (t_r[\mathit{fidx}]^*, t_g[\mathit{fidx}]^*)
\end{aligned}
$$

The instrumentation of helper functions is given below. Unary and binary operations propagate the information from their parameters to their result. Function zeros^T provides a sequence of zeros with no taint. Function taint-params (resp. taint-globals) marks parameters (resp. global variables) with their taint.

$$
\begin{aligned}
\mathsf{unary}^T(\mathit{unop}, (v, t)) &= (\mathsf{unary}(\mathit{unop}, v), t) \\
\mathsf{binary}^T(\mathit{binop}, (v_1, t_1), (v_2, t_2)) &= (\mathsf{binary}(\mathit{binop}, v_1, v_2), t_1 \sqcup t_2) \\
\mathsf{zeros}^T(0) &= \epsilon \\
\mathsf{zeros}^T(n) &= (0, \bot) \cdot \mathsf{zeros}^T(n-1) \\
\mathsf{taint\text{-}params}(\mathit{fidx}, \epsilon, n) &= \epsilon \\
\mathsf{taint\text{-}params}(\mathit{fidx}, (v, t) \cdot V, n) &= (v, t \sqcup [\mathit{fidx} \mapsto \{\mathbf{parameter}(n)\}]) \\
&\quad \cdot \mathsf{taint\text{-}params}(\mathit{fidx}, V, n+1) \\
\mathsf{taint\text{-}globals}(\mathit{fidx}, \epsilon, n) &= \epsilon \\
\mathsf{taint\text{-}globals}(\mathit{fidx}, (v, t) \cdot G, n) &= (v, t \sqcup [\mathit{fidx} \mapsto \{\mathbf{global}(n)\}]) \\
&\quad \cdot \mathsf{taint\text{-}globals}(\mathit{fidx}, G, n+1)
\end{aligned}
$$

Finally, the instrumentation of the semantics of MiniWasm is given in Figure 2, where the instrumentation is highlighted in grey. Only the non-trivial cases are given. The states are extended with a set of summaries produced, S. The `i32.const` instruction pushes a new value that has no taint. Unary (resp. binary) operations propagate information flow according to the unary^T (resp. binary^T) helper functions. Upon a function invocation, the parameters and global variables are marked with the appropriate taint, while other values have no taint.

Upon a function return, the summary of the corresponding function call is added to the set of summaries.

With this instrumented semantics, the execution of a module produces a set of summaries, where each summary corresponds to information flow for one function execution.

V. Compositional Information Flow Analysis

We now turn to the static approximation of information flow function summaries. As motivated in Section II, the static analysis is *compositional*: one function is analysed at a time, independently of its callers. Once a function summary has been inferred, it can subsequently be used to produce summaries for the callers of a function.

A. Control Flow Graphs

The information flow analysis we describe here is expressed as a data flow analysis on a control flow graph (CFG). A CFG is a set of basic blocks connected by edges, with a single entry block and a single exit block. Each MiniWasm basic block either contains one control instruction, or is a sequence of data instructions. Jumps occur from control instructions to the start of another block. The CFG of a MiniWasm function can be constructed using the traditional CFG construction approach [10]. Unlike analyses for other binary instruction formats [14], [23], [35], [45], [54], analyses for WebAssembly do not require approximating control flow jumps: the target of jump instructions (`br`, `br_if`, `br_table`) is always explicit.

B. Runtime Structure Inference

The information flow analysis relies on the ability to identify and name elements of the stack, locals, and globals. To that end, the analysis performs a first pass over the CFG to infer the shape of the stack and to assign a unique name for each stack location, local variable and global variable. After this inference phase, the size of the value stack before and after each instruction is known precisely. The following example is annotated with the inferred names for the stack, the locals, and the globals.

```
(type (func i32 -> i32))
(func (type 1) (local i32)
  ;; stack: [], locals: [p0,l0], globals: [g0]
  i32.const 1
  ;; stack: [i0], locals: [p0,l0], globals: [g0]
  local.get 0
  ;; stack: [i1,i0], locals: [p0,l0], globals: [g0]
  i32.add)
  ;; stack: [i2], locals: [p0,l0], globals: [g0]
```

This analysis phase is performed as a walk through the CFG. For each instruction, we statically know how it modifies the stack. For example, the `i32.const 1` instruction pushes a new value on the stack, hence a new name (`i0`) is created for that new stack location. The process is similar for instructions manipulating local and global variables, e.g., `local.set` updates a local variable, which is therefore assigned a new name. Special care needs to be taken upon merge points in the CFG: names that may differ have to be replaced. To do so, we extend the CFG with an extra merge node at every merge of the control flow. In this merge node, all names are replaced by fresh names. For example, if two nodes are connected to a merge node, where the top value of the stack has name x in one node, and name y in the other node, the top of the stack at the merge node will be z, and the analysis has to account that z results from joining x and y. This is similar to the use of ϕ-nodes in compilers [50]. It is up to the analysis to correctly deal with these names.

C. Information Flow Analysis

The information flow analysis computes, before and after each instruction, a map of names to the information that they may contain, represented as a set of *Taint*.

$$S \in \mathit{State} = \mathit{Name} \rightarrow \mathcal{P}(\mathit{Taint})$$

The initial state of the analysis for a function that has n parameters, in a module that has m global variables, is the following, where all parameters and globals are assigned their own taint. We assume p_i (resp. g_i) is the name corresponding to the ith parameter (resp. global variable).

$$\begin{aligned} S_0 = [&p_0 \mapsto \{\textbf{parameter}(0)\}, \dots p_n \mapsto \{\textbf{parameter}(n)\},\\ &g_0 \mapsto \{\textbf{global}(0)\} \dots g_m \mapsto \{\textbf{global}(m)\}] \end{aligned}$$

The analysis is then described as a state transformer for each instruction: $[\![\mathit{instr}]\!](S, V, L, G, V', L', G')$, defined in Figure 3. This state transformer computes the state after the instruction, given the state before the instruction (S), and using information from the runtime structure inference phase, namely the names of the value stack, locals, and globals before the instruction (V, L, and G), as well as after the instruction (V', L', and G').

Instructions like `drop` and `i32.const` do not propagate any information. Unary and binary operations propagate information from their parameters to the resulting value. Instructions to manipulate locals and globals propagate information as follows. After getting the value of a local (resp. global) with `local.get` (resp. `global.get`), the top value of the stack is tainted with the taint from the local (resp. global). Modifying the value of a local (resp. global) with `local.set` (resp. `global.set`) propagates the information from the value set to the resulting local (resp. global).

For a `store` instruction, information is propagated from the stored value to the special name `mem`, which indicates that information may flow anywhere in the memory. This is a deliberately sound but coarse modelling of the memory, which cannot be refined without a precise modelling of numerical values. We leave such refinements for future work. For a `load` instruction, information is propagated from the special name `mem` to the resulting value.

For a `call` n instruction, the function that is called is known precisely: it is the nth function of the module. The information is propagated according to the information flow summary of that function, which we will describe shortly. For a `call_indirect` t instruction, the called function is not known precisely, because the analysis does not derive any numerical properties, and the function called depends on the top value of the stack. However, the type t of the called

$$
\begin{aligned}
V, \mathbf{plain}(\texttt{i32.const}\ n) \cdot \mathit{as}, L, G, M, \colorbox{lightgray}{S} &\rightarrow (n, \colorbox{lightgray}{$\bot$}) \cdot V, \mathit{as}, L, G, M, \colorbox{lightgray}{S} \\
v_2 \cdot v_1 \cdot V, \mathbf{plain}(\texttt{i32.}\mathit{binop}) \cdot \mathit{as}, L, G, M, \colorbox{lightgray}{S} &\rightarrow \colorbox{lightgray}{binary^T} (\mathit{binop}, v_1, v_2) \cdot V, \mathit{as}, L, G, M, \colorbox{lightgray}{S} \\
v \cdot V, \mathbf{plain}(\texttt{i32.}\mathit{unop}) \cdot \mathit{as}, L, G, M, \colorbox{lightgray}{S} &\rightarrow \colorbox{lightgray}{unary^T} (\mathit{unop}, v) \cdot V, \mathit{as}, L, G, M, \colorbox{lightgray}{S} \\
V, \mathbf{invoke}((\mathit{ft}, n, \mathit{instr}^*), \mathit{fidx}) \cdot \mathit{as}, L, G, M, \colorbox{lightgray}{S} &\rightarrow V, \mathbf{frame}(\mathit{ft}_{\mathit{out}}, L', \epsilon, \mathbf{label}(\mathit{ft}_{\mathit{out}}, \epsilon, \epsilon, \mathbf{plain}(\mathit{instr})^*), \mathit{fidx}) \cdot \mathit{as}, L, \colorbox{lightgray}{G'}, M, \colorbox{lightgray}{S} \\
\text{where } L' &= \colorbox{lightgray}{$\mathsf{taint\text{-}params}(\mathit{fidx}, \mathsf{rev}(\mathsf{take}(\mathit{ft}_{\mathit{in}}, V), 0)) \cdot \mathsf{zeros}^T(n)$} \\
G' &= \colorbox{lightgray}{$\mathsf{taint\text{-}globals}(\mathit{fidx}, G, 0)$} \\
V, \mathbf{frame}(n, \_, V', \epsilon, \mathit{fidx}) \cdot \mathit{as}, L, G, M, \colorbox{lightgray}{S} &\rightarrow V' \cdot V, \mathit{as}, L, G, M, \colorbox{lightgray}{$\{\mathsf{summary}(\mathit{fidx}, V', G)\} \cup S$}
\end{aligned}
$$

Fig. 2. Instrumented semantics of MiniWasm.

$$
\begin{aligned}
\llbracket \texttt{drop} \rrbracket(S, V, L, G, V', L', G') &= S &\qquad \llbracket \texttt{local.set}\ n \rrbracket(S, v \cdot V, L, G, V', L', G') &= S[L'_n : S[v]] \\
\llbracket \texttt{i32.const} \rrbracket(S, V, L, G, V', L', G') &= S & \llbracket \texttt{global.get}\ n \rrbracket(S, V, L, G, v' \cdot V', L', G') &= S[v' : S[G_n]] \\
\llbracket \texttt{i32.}\mathit{binop} \rrbracket(S, v_2 \cdot v_1 \cdot V, L, G, v' \cdot V', L, G) &= S[v' : S[v_1] \sqcup S[v_2]] & \llbracket \texttt{global.set}\ n \rrbracket(S, v \cdot V, L, G, V', L', G') &= S[G'_n : S[v]] \\
\llbracket \texttt{i32.}\mathit{unop} \rrbracket(S, v \cdot V, L, G, v' \cdot V', L', G') &= S[v' : S[v]] & \llbracket \texttt{store} \rrbracket(S, v_2 \cdot v_1 \cdot V, L, G, V', L', G') &= S[\mathsf{mem} : S[v_1]] \\
\llbracket \texttt{local.get}\ n \rrbracket(S, V, L, G, v' \cdot V', L', G') &= S[v' : S[L_n]] & \llbracket \texttt{load} \rrbracket(S, V, L, G, v' \cdot V', L', G') &= S[v' : S[\mathsf{mem}]]
\end{aligned}
$$

$$
\begin{aligned}
\llbracket \texttt{call}\ n \rrbracket(S, V, L, G, V', L', G') &= \mathsf{apply\text{-}summary}(n, S, \mathsf{rev}(\mathsf{take}(\mathit{ft}_{\mathit{in}}, V)), G, \mathit{vs'}, G') \\
\llbracket \texttt{call\_indirect}\ t \rrbracket(S, v \cdot V, L, G, V', L', G') &= \bigsqcup_{n \in \mathsf{matching\text{-}funs}(t)} \mathsf{apply\text{-}summary}(n, S, \mathsf{rev}(\mathsf{take}(\mathit{ft}_{\mathit{in}}, V)), G, V', G')
\end{aligned}
$$

Fig. 3. State transformer for information flow analysis.

function is known. Hence, we know that a function that is the target of the call has to have a matching type, and has to be declared in the table of the module. We can therefore statically compute a set of functions that may be called, which is what we assume $\mathsf{matching\text{-}funs} : \mathit{Type} \rightarrow \mathcal{P}(\mathbb{N})$ does, returning a set of function indices. Then, each of these functions' summaries are applied, joining the results together.

D. Information Flow Summaries

An information flow summary describes how information propagates from a function's parameters and globals to its optional return value and to the globals after its execution. We present these summaries formally, for the case where there is exactly one return value. The general case is similar. Auxiliary helper function $\mathsf{mk\text{-}summary}$ constructs a summary from the state S of the information flow analysis at the end of the function, where v is the name of the top of the stack, and G are the names of the global variables at the end of the function execution. The bottom summary $\bot$ maps the return value and each global to the empty set.

$$
\begin{aligned}
\mathit{Summary} &= \mathcal{P}(\mathit{Taint}) \times \mathcal{P}(\mathit{Taint})^* \\
\mathsf{mk\text{-}summary}(S, v, G) &= (S[v], S[G_1] \cdots S[G_m]) \\
\bot &= (\varnothing, \varnothing \cdots \varnothing)
\end{aligned}
$$

Function $\mathsf{apply\text{-}summary}$ applies a summary by first constructing a substitution σ, that will replace occurrences of parameters and globals in a set of taints by their taint before the call. Then, it sets the taint of the return value v' to the taint given by the summary after substitution, and similarly for all global variables. The summary itself is extracted by $\mathsf{lookup\text{-}summary}$, which provides the summary for function n.

$$
\begin{aligned}
&\mathsf{apply\text{-}summary}(n, S, V, G, v' \cdot V', G') \\
&\qquad = S[v' \mapsto \sigma(r), G'_0 \mapsto \sigma(G''_0), \ldots, G'_m \mapsto \sigma(G''_m)] \\
&\text{where } (r, G'') = \mathsf{lookup\text{-}summary}(n) \\
&\qquad \sigma(X) = \bigsqcup_{\mathbf{parameter}(i) \in X} S[V_i] \ \sqcup \bigsqcup_{\mathbf{global}(i) \in X} S[G_i]
\end{aligned}
$$

E. Intra-Procedural Analysis

The state transformer presented in Figure 3 can be computed for all instructions of a function, following the traditional dataflow analysis approach [47]. The only special case to take into account is to handle merge nodes of the CFG, which were introduced by the runtime structure inference. For our taint analysis, in case of a merge node that merges a variable x with a variable y into a variable z, the taint of z is $S[x] \sqcup S[y]$.

F. Bottom-Up Inter-Procedural Analysis

We now have described a compositional analysis for MiniWasm functions. The main remaining question is: how do we know the summary to apply upon a function call? This is solved by performing a bottom-up inter-procedural analysis.

Computing the call graph. First, a call graph has to be computed for the WebAssembly module. Because all function call targets are either statically known in the case of the `call` instruction, or can be approximated by their type and the content of the module table in the case of the `call_indirect` instruction, we can derive an approximate call graph.

Computing the analysis schedule. From this approximate call graph, we apply Tarjan's algorithm [62] to compute the strongly-connected components (SCCs) of the call graph, in topological order. The order of the SCCs gives us an *analysis schedule*: an SCC x precedes SCC y in this sequence if a function from y may call a function from x, hence x has to be analysed before y so that its summary is available during the analysis of y.

Analysing an SCC. The analysis of the entire module is therefore decomposed into the analysis of a set of functions that form an SCC. Each function of the SCC can then be analysed by the compositional analysis. A function call within an SCC either calls a function that has been fully analysed, as part of a previous SCC, or calls a function within the same SCC. In the first case, the summary of the called function is fully known and can be used. In the second case, the summary is unknown and we rely on the bottom summary. When one function of the SCC has been analysed, its callers that are within the same SCC are scheduled for (re-)analysis, as they can now rely on a more complete summary. This proceeds in a fixed-point fashion, until all summaries have reached their fixed point, and the analysis can proceed with the next SCC.

Handling Imported Functions. A MiniWasm module can have functions that are not defined, but that rather are *imported*. In WebAssembly, such functions usually stem from WASI. For each imported function, we manually encode an information flow summary. For example, WASI's `proc_exit` function takes an argument and terminates the execution of the program. Hence, its summary is the bottom summary: it does not propagate any information.

G. *Soundiness of Summaries*

A summary does not retain information about how information flows within the memory during the function execution. This is a deliberate choice that breaks the soundness of the analysis at the benefit of precision. This similar to what the *soundiness manifesto* advocates [39]. As we shall see in our evaluation, we did not encounter any unsound result despite this choice: WebAssembly functions do not seem to store values that are coming from memory in the global variables, nor to return such values. It would be possible to retain this information by preserving the information from the special name `mem` in the summary, and propagate it upon application. This may cause a precision loss though: any function that stores one of its arguments in memory taints the entire memory with the information from its argument. Subsequent loads in this function would be tainted with that information. We envision that retaining information about numerical values would allow a more precise modelling of the store, and hence more precise summaries even when the store is included.

VI. Evaluation

We have implemented the information flow analysis presented here in around 3000 LOC of OCaml code.[1] Our implementation is not limited to MiniWasm, but supports the complete WebAssembly standard. To assess the precision of the results of the analysis, we also instrumented the concrete interpreter that accompanies the WebAssembly specification [9], following our instrumentation presented in Section IV.

We have run our analysis on 34 C programs, coming from two benchmark sources: the first 30 are the entirety of the PolyBench benchmarks [48] which implement arithmetic kernels, and all feature indirect function calls. The remaining 4 are selected from the Language Benchmark Game [26], which aim at evaluating performance of programming languages, and do not require complex language features such as parallelism, nor indirect function calls. Both have been used in the evaluation of program analyses [15], [19], [46], [53]. We compiled each program to WebAssembly using `clang` 10.0.1, and linked with a `libc` implementation built on top of WASI [2]. We used a laptop with an Intel i7-8650U CPU, with 32GB of RAM, with OCaml 4.10.0.

TABLE I
Benchmark programs, analysis time (in seconds) and precision of resulting summaries (Prec.).

Program	**LOC**	**Time**	**Prec.**	**Program**	**LOC**	**Time**	**Prec.**
2mm	6815	2.09	6/9	heat-3d	5935	1.47	4/8
3mm	6906	2.09	6/9	jacobi-1d	5708	1.44	4/8
adi	6002	1.48	4/8	jacobi-2d	5811	1.46	4/8
atax	6777	2.07	7/10	lu	6164	1.53	5/9
bicg	6779	2.05	7/10	ludcmp	7202	2.18	7/10
cholesky	6142	1.52	5/9	mvt	6663	2.04	6/9
correlation	6769	2.07	6/9	nussinov	6809	2.11	6/10
covariance	6672	2.07	6/9	seidel-2d	5756	1.42	4/8
deriche	6052	1.52	4/8	symm	6771	2.04	6/9
doitgen	6681	1.97	6/9	syr2k	6696	2.02	6/9
durbin	5762	1.51	4/8	syrk	6642	2.02	6/9
fdtd-2d	6760	1.99	6/9	trisolv	6590	2.10	6/9
floyd-warshall	5760	1.38	4/8	trmm	6649	2.03	6/9
gemm	6685	2.07	6/9	fankuchredux	439	0.08	4/4
gemver	6778	2.05	6/9	mandelbrot	945	0.07	2/2
gesummv	6632	2.00	6/9	nbody	295	0.03	2/2
gramschmidt	6889	2.10	6/9	spectral-norm	221	0.06	3/3

To measure precision, we ran both the static analysis and the instrumented concrete interpreter (i.e., a dynamic analysis) in order to generate static and dynamic information flow summaries respectively. For each function, we evaluate the precision of the summary S_s produced by the static analysis as follows. First, we join all dynamic summaries $S_{d_1}, \dots, S_{d_n}$ produced for that function (each dynamic summary correspond to one function execution), resulting in S_d. In case no summary has been produced by the dynamic analysis, this means that the function was not reached by the dynamic analysis. We therefore cannot assess the precision of the static analysis for that function, and we ignore these functions in our evaluation. For functions that have dynamic summaries, we have one of the following situations:

- $S_d = S_s$, in which case the static analysis is fully precise.
- S_s misses information from S_d, in which case the static analysis has produced an unsound result (i.e., a false negative). This did not happen in our evaluation, even though summaries are *soundy* [39] as they do not model the taint of the linear memory.
- S_s includes more information than S_d, e.g., $S_s = (\{\textbf{parameter}(0)\}, \epsilon)$ (the return value contains information

[1] Available at: https://github.com/acieroid/wassail/releases/tag/scam2020

from its first parameter) and $S_d = (\varnothing, \epsilon)$ (the return value does not contain any information from its parameters). This means that the static analysis *may* have produced imprecise results for that function (i.e., a false positive). We therefore count this as an imprecise summary. In practice, however, it can be that some paths within a function that are not reached by the dynamic analysis, are taken into account by the static analysis. In this case, even though the summary derived by the static analysis may be fully precise, it will not be reported as so. Hence, the precision reported here is a lower bound on the actual precision of the analysis.

Computing this for all functions of a module gives us a ratio, e.g., 6/9 for the 2mm program, indicating that the static analysis computed 6 static summaries that are fully precise and are true positives, and 3 that are potentially false positives. This evaluation method is similar to how other taint analysers are evaluated [49], with the difference that we measure precision on function summaries rather than solely on unwanted information flows.

Table I lists the benchmark programs along with the time taken to analyse them, and our measure of precision. In total, 196157 LOCs are analysed in 56.13 seconds, with a precision lower bound of 64% for the resulting summaries. The analysis running time is low: each benchmarks is analysed within 2 seconds, and on average 3495 LOC are analysed per second, with peaks up to 13500 LOC/s for shorter benchmarks like `mandelbrot`. The total precision is of 64%, which is in line with other static taint analysis tools [49], [64]. We notice however that the precision varies depending on the benchmark suite: it is of 100% on programs from the Language Benchmark Game, and of 62% on programs from the PolyBench benchmark suite. A closer look at the results for the PolyBench benchmark suite shows that many of the programs share common functions, some of which are analysed imprecisely, which propagates to all analysis results.

VII. Related work

A. WebAssembly

Even though WebAssembly is a recent standard, there has been some work focusing on its analysis and verification. Lehmann et al. [38] present a dynamic analysis framework for WebAssembly. Instead of instrumenting a WebAssembly interpreter (Section IV), one could instrument WebAssembly programs with Wasabi to compute the same information. Watt et al. [69] propose a separation logic for WebAssembly, enabling manual verification of functions and modules. Our approach derives less expressive properties, but in a fully automated way. Another existing approach, taken by CT-Wasm, is to provide a rich type system for WebAssembly [70].

B. Summary-Based Analyses

Summary-based analyses have been used on numerous occasions to achieve scalable static analyses, as an alternative to whole-program analyses. Saturn [11] is summary-based, using logic programming to analyse C programs. The design of summaries is left to the analysis designer. A points-to analysis developed with Saturn can scale up to the size of the Linux kernel. Tang et al. [61] present summaries in the presence of callbacks in libraries, when a function may have to be reanalysed even if a summary has already been produced, due to new reachability relations. This is not a situation that can occur in our case as each function is analysed independently of its callers. Cassez et al. [18] demonstrate the use of function summaries in a modular analysis, using trace abstraction refinement. Yan et al. [72] propose to redesign the Soot framework for supporting summaries, arguing that summaries should be integrated within program analysis frameworks.

C. Taint Analysis

A taint analysis identifies information flow from a *source* (e.g., user input) to a *sink* (e.g., a database query). There exists general-purpose static taint analyses [63], [64] and specific static taint analyses, for Android [13], [36], web applications [34], JavaScript [57], and event-driven programs [21].

The notion of *taint summary* is not new. Zhang et al. [73] rely on taint summaries to optimise a dynamic taint analysis. Staicu et al. [57] dynamically extract *taint specifications* for JavaScript libraries, which can then be used by static analysis tools. The taint specifications inferred are more specific than our summaries due to the object-oriented nature of JavaScript.

D. Static Analysis of Binaries

There is extensive work on static analysis of binaries for different platforms. We already covered existing work for WebAssembly, and we cover here work on different platforms, that are the most related to this work. Most importantly, Ballabriga et al. [15] recently presented a static analysis that focuses on memory indirections by the use of polyhedral numerical domains, applied to ARM binaries. Adapting this to our setting would allow the analysis to properly support memory indirections. Moreover, the use of numeric domains as done by related work on binary analysis [42], [54] could improve the precision of our analysis: knowledge of numerical properties would enable resolving indirect calls with more precision, as well as a better modelling of the memory. The Soot framework [65] analyses JVM bytecode, which is also a stack-based binary format. Our runtime structure inference shares similarities with Soot's analysis phases on Baf bytecode [66], namely that the shape of the stack needs to be inferred before and after each instruction.

VIII. Conclusion

The WebAssembly standard is gaining popularity, and there is a need for analysis tools to assess the quality of WebAssembly modules. In this paper, we propose the first compositional static analysis for WebAssembly, in the form of an information flow analysis. This analysis is compositional, enabling it to analyse functions independently of their calling context, resulting in a scalable analysis.

Acknowledgements

This work was partially supported by the *"Cybersecurity Initiative Flanders"*.

References

[1] Lucet, the sandboxing webassembly compiler. https://github.com/bytecodealliance/lucet.

[2] WASI libc implementation for webassembly. https://github.com/WebAssembly/wasi-libc.

[3] Wasm3: A high performance webassembly interpreter written in c. https://github.com/wasm3/wasm3.

[4] Wasmer: The leading webassembly runtime supporting wasi and emscripten. https://github.com/wasmerio/wasmer.

[5] WasmTime: A standalone runtime for webassembly. https://github.com/bytecodealliance/wasmtime.

[6] WebAssembly. https://webassembly.org/.

[7] WebAssembly Core Specification. URL: https://www.w3.org/TR/wasm-core-1/.

[8] WebAssembly: Security. https://webassembly.org/docs/security/.

[9] Webassembly specification, reference interpreter, and test suite. https://github.com/WebAssembly/spec.

[10] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[11] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. An overview of the saturn project. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'07, San Diego, California, USA, June 13-14, 2007*, pages 43–48, 2007.

[12] Bytecode Alliance. WASI: The webassembly system interface. https://wasi.dev/.

[13] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269, 2014.

[14] Gogul Balakrishnan and Thomas W. Reps. Analyzing memory accesses in x86 executables. In *Compiler Construction, 13th International Conference, CC 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, pages 5–23, 2004.

[15] Clément Ballabriga, Julien Forget, Laure Gonnord, Giuseppe Lipari, and Jordy Ruiz. Static analysis of binary code with memory indirections using polyhedra. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*, pages 114–135, 2019.

[16] Weikang Bian, Wei Meng, and Yi Wang. Poster: Detecting webassembly-based cryptocurrency mining. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 2685–2687, 2019.

[17] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. Racerd: compositional static race detection. *Proc. ACM Program. Lang.*, 2(OOPSLA):144:1–144:28, 2018.

[18] Franck Cassez, Christian Müller, and Karla Burnett. Summary-based inter-procedural analysis via modular trace refinement. In *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India*, pages 545–556, 2014.

[19] Dong Chen, Fangzhou Liu, Chen Ding, and Sreepathi Pai. Locality analysis through static parallel sampling. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 557–570, 2018.

[20] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, pages 159–178, 2002.

[21] Jonas De Bleser, Quentin Stiévenart, Jens Nicolay, and Coen De Roover. Static taint analysis of event-driven scheme programs. In *Proceedings of the 10th European Lisp Symposium (ELS 2017), Brussels, Belgium, April 3-4, 2017*, pages 80–87, 2017.

[22] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. Position paper: Progressive memory safety for webassembly. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy, HASP@ISCA 2019, June 23, 2019*, pages 4:1–4:8, 2019.

[23] Adel Djoudi and Sébastien Bardin. BINSEC: binary code analysis with low-level regions. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, 2015, London, UK, April 11-18, 2015. Proceedings*, pages 212–217, 2015.

[24] Azadeh Farzan and Zachary Kincaid. Compositional bitvector analysis for concurrent programs with nested locks. In *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, pages 253–270, 2010.

[25] William Fu, Raymond Lin, and Daniel Inge. Taintassembly: Taint-based information flow control tracking for webassembly. *CoRR*, abs/1802.01050, 2018.

[26] Brent Fulgham and Isaac Gouy. The computer language benchmarks game. https://benchmarksgame-team.pages.debian.net/benchmarksgame/.

[27] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 43–56, 2010.

[28] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. Acctee: A webassembly-based two-way sandbox for trusted resource accounting. In *Proceedings of the 20th International Middleware Conference, Middleware 2019, Davis, CA, USA, December 9-13, 2019*, pages 123–135, 2019.

[29] Eric Goubault, Sylvie Putot, and Franck Védrine. Modular static analysis with zonotopes. In *Static Analysis - 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, pages 24–40, 2012.

[30] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Shooting from the heap: ultra-scalable static analysis with heap snapshots. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 198–208, 2018.

[31] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 185–200, 2017.

[32] Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation, IoTDI 2019, Montreal, QC, Canada, April 15-18, 2019*, pages 225–236, 2019.

[33] Matthieu Journault, Antoine Miné, and Abdelraouf Ouadjaout. Modular static analysis of string manipulations in C programs. In *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, pages 243–262, 2018.

[34] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Static analysis for detecting taint-style vulnerabilities in web applications. *J. Comput. Secur.*, 18(5):861–907, 2010.

[35] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, pages 423–427, 2008.

[36] William Klieber, Lori Flynn, Will Snavely, and Michael Zheng. Practical precise taint-flow static analysis for android app sets. In *Proceedings of the 13th International Conference on Availability, Reliability and Security, ARES 2018, Hamburg, Germany, August 27-30, 2018*, pages 56:1–56:7, 2018.

[37] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of webassembly. In *29th USENIX Security Symposium, USENIX Security 2020, Virtual, August 12-14, 2020*, 2020.

[38] Daniel Lehmann and Michael Pradel. Wasabi: A framework for dynamically analyzing webassembly. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, pages 1045–1058, 2019.

[39] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z Guyer, Uday P Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundness: a manifesto. *Communications of the ACM*, 58(2):44–46, 2015.

[40] Brian McFadden, Tyler Lukasiewicz, Jeff Dileo, and Justin Engler. Security chasms of wasm, 2018.

[41] Xiaozhu Meng and Barton P. Miller. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 24–35, 2016.

[42] Antoine Miné. Abstract domains for bit-level machine integer and floating-point operations. In *ATx'12/WInG'12: Joint Proceedings of the Workshops on Automated Theory eXploration and on Invariant Generation, Manchester, UK, June 2012*, pages 55–70, 2012.

[43] Antoine Miné. Relational thread-modular static value analysis by abstract interpretation. In *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, pages 39–58, 2014.

[44] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. New kid on the web: A study on the prevalence of webassembly in the wild. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19-20, 2019, Proceedings*, pages 23–42, 2019.

[45] Minh Hai Nguyen, Thien Binh Nguyen, Thanh Tho Quan, and Mizuhito Ogawa. A hybrid approach for control flow graph construction from binary code. In *20th Asia-Pacific Software Engineering Conference, APSEC 2013, Ratchathewi, Bangkok, Thailand, December 2-5, 2013 - Volume 2*, pages 159–164, 2013.

[46] Jens Nicolay, Quentin Stiévenart, Wolfgang De Meuter, and Coen De Roover. Effect-driven flow analysis. In *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*, pages 247–274, 2019.

[47] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999.

[48] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. https://web.cse.ohio-state.edu/~pouchet.2/software/polybench.

[49] Lina Qiu, Yingying Wang, and Julia Rubin. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 176–186, 2018.

[50] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 12–27, 1988.

[51] Salim S. Salim, Andy Nisbet, and Mikel Luján. Towards a webassembly standalone runtime on graalvm. In *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH 2019, Athens, Greece, October 20-25, 2019*, pages 15–16, 2019.

[52] Salim S. Salim, Andy Nisbet, and Mikel Luján. Trufflewasm: a webassembly interpreter on graalvm. In *VEE '20: 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, virtual event, Lausanne, Switzerland, March 17, 2020*, pages 88–100, 2020.

[53] Gülfem Savrun-Yeniçeri, Wei Zhang, Huahan Zhang, Chen Li, Stefan Brunthaler, Per Larsen, and Michael Franz. Efficient interpreter optimizations for the JVM. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Stuttgart, Germany, September 11-13, 2013*, pages 113–123, 2013.

[54] Alexander Sepp, Bogdan Mihaila, and Axel Simon. Precise static analysis of binaries by extracting relational information. In *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*, pages 357–366, 2011.

[55] Robbert Gurdeep Singh and Christophe Scholliers. Warduino: a dynamic webassembly virtual machine for programming microcontrollers. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2019, Athens, Greece, October 21-22, 2019*, pages 27–36, 2019.

[56] Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei Sabelfeld. An empirical study of information flows in real-world javascript. In *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, 2019.

[57] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. Extracting taint specifications for javascript libraries. In *Proc. 42nd International Conference on Software Engineering (ICSE)*, 2020.

[58] Quentin Stiévenart, Jens Nicolay, Wolfgang De Meuter, and Coen De Roover. A general method for rendering static analyses for diverse concurrency models modular. *J. Syst. Softw.*, 147:17–45, 2019.

[59] Jian Sun, DingYuan Cao, Ximing Liu, ZiYi Zhao, WenWen Wang, XiaoLi Gong, and Jin Zhang. Selwasm: A code protection mechanism for webassembly. In *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking, ISPA/BDCloud/SocialCom/SustainCom 2019, Xiamen, China, December 16-18, 2019*, pages 1099–1106, 2019.

[60] Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. Taint tracking for webassembly. *CoRR*, abs/1807.08349, 2018.

[61] Hao Tang, Xiaoyin Wang, Lingming Zhang, Bing Xie, Lu Zhang, and Hong Mei. Summary-based context-sensitive data-dependence analysis in presence of callbacks. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 83–95, 2015.

[62] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[63] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 210–225, 2013.

[64] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 87–97, 2009.

[65] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, page 13, 1999.

[66] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Compiler Construction, 9th International Conference, CC 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings*, pages 18–34, 2000.

[67] Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W. Hamlen, and Shuang Hao. SEISMIC: secure in-lined script monitors for interrupting cryptojacks. In *Computer Security - 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part II*, pages 122–142, 2018.

[68] Conrad Watt. Mechanising and verifying the webassembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 53–65, 2018.

[69] Conrad Watt, Petar Maksimovic, Neelakantan R. Krishnaswami, and Philippa Gardner. A program logic for first-order encapsulated webassembly. In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom*, pages 9:1–9:30, 2019.

[70] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. Ct-wasm: type-driven secure cryptography for the web ecosystem. *Proc. ACM Program. Lang.*, 3(POPL):77:1–77:29, 2019.

[71] John Whaley and Martin C. Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999*, pages 187–206, 1999.

[72] Dacong Yan, Guoqing (Harry) Xu, and Atanas Rountev. Rethinking soot for summary-based whole-program analysis. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, SOAP 2012, Beijing, China, June 14, 2012*, pages 9–14, 2012.

[73] Ruoyu Zhang, Shan Huang, and Zhengwei Qi. Efficient taint analysis with taint behavior summary. In *Third International Conference on Communications and Mobile Computing, CMC 2011, Qingdao, China, 18-20 April 2011*, pages 11–14, 2011.