Building Call Graph of WebAssembly Programs via Abstract Semantics

Mattia Paccamiccio[1,*], Franco Raimondi[2] and Michele Loreti[1]

[1]Università di Camerino, Via Andrea d'Accorso 16, 62032, Camerino, Italy
[2]Gran Sasso Science Institute, Viale Francesco Crispi 7, 67100, L'Aquila, Italy

Abstract

WebAssembly is a binary format for code that is gaining popularity thanks to its focus on portability and performance. Currently, the most common use case for WebAssembly is execution in a browser. It is also being increasingly adopted as a stand-alone application due to its portability. The binary format of WebAssembly, however, makes it prone to being used as a vehicle for malicious software. For instance, one could embed a cryptocurrency miner in code executed by a browser. As a result, there is substantial interest in developing tools for WebAssembly security verification, information flow control, and, more generally, for verifying behavioral properties such as correct API usage. In this document, we address the issue of building call graphs for WebAssembly code. This is important because having or computing a call graph is a prerequisite for most inter-procedural verification tasks. In this paper, we propose a formal solution based on the theory of Abstract Interpretation. We compare our approach to the state-of-the-art by predicting how it would perform against a set of specifically crafted benchmark programs.

Keywords

WebAssembly, Call Graph, Abstract Interpretation, Semantics, Program Analysis

1. Introduction

WebAssembly *"is a binary instruction format for a stack-based virtual machine, designed as a portable compilation target for programming languages"*[1]. WebAssembly aims to be portable and efficient: it is actively supported by most web browsers and it is employed for rendering streamed videos on devices such as smart televisions, set-top boxes, and other embedded devices [1, 2].

WebAssembly is primarily used in conjunction with JavaScript to optimize tasks that are typically resource-intensive and would benefit from more optimized, close-to-native performance [1], such as rendering images, streaming videos and generally heavy computational tasks for which interpreted languages might be too slow.

In parallel with its adoption, researchers have investigated the issue of verifying WebAssembly code directly [3]. From a security point of view, this is motivated by the fact that WebAssembly, while executing in a relatively safe sandboxed environment, is still the source of security

ITASEC 2024: The Italian Conference on CyberSecurity, April 08-12, 2024, Salerno, Italy

[*] Corresponding author.

mattia.paccamiccio@unicam.it (M. Paccamiccio); f.raimondi@gssi.it (F. Raimondi); michele.loreti@unicam.it (M. Loreti)

0000-0003-2096-0191 (M. Paccamiccio); 0000-0002-9508-7713 (F. Raimondi); 0000-0003-3061-863X (M. Loreti)

[1] https://webassembly.org/

concerns (information leakage, tampering, policy violation, unauthorized use of computational power, etc.).

In terms of formal analysis, a call graph is a prerequisite for verifying several properties of interest, such as information flow analysis, call-ordering requirements, safety, and liveness. We define a call graph as a directed graph that encodes the possible calling relationships between functions or methods in a program, represented as nodes, and edges between two nodes (i.e. f, g) representing that f calls g. Constructing an *exact* call graph is an undecidable problem. The most adopted approach is soundly approximating all the possible calls, implying that an edge between f and g, in a sound approximation, translates to f *may* call g. The precision of a call graph depends on the inference rules used, which could include or ignore information about execution paths, context, and so on.

1.1. Contribution

We investigate the problem of building a precise call graph for WebAssembly. The simple execution model of WebAssembly, briefly described in Section 2, implies the need to track values on top of the stack to resolve call sites relying on some form of indirection. Other sound approaches can be improved in precision [4, 5], while precise approaches are unsound (MetaDCE[2], WAVM[3], Manticore [6]).

We propose an approach based on the theory of Abstract interpretation, aiming to improve precision while retaining soundness.

We lay out the paper as follows: we introduce background details about WebAssembly and Abstract interpretation in Section 2; we describe our formal approach in Section 3; we present related work in Section 4 and we conclude in Section 5.

2. Background

In this section, we start by giving an overview of WebAssembly. We then introduce basic notions of call graphs, what soundness entails when analyzing WebAssembly modules, and Abstract interpretation.

2.1. Basics of WebAssembly

WebAssembly is a simple, typed, assembly-like language whose execution model is based on a stack machine, implementing a set of low-level instructions, less than 200 in total. It is designed to be a universal compilation target, mostly for the LLVM family of programming languages. The resulting binary is then run on an ad-hoc Virtual Machine (VM). The most common example of WebAssembly Virtual Machine as of today is a sandboxed environment in a browser, but stand-alone runtime environments for resource-constrained devices are available[4]. Overall, the paradigm is very similar to the one that Java adopts: source code is compiled into bytecode and

[2] https://github.com/WebAssembly/binaryen

[3] https://wavm.github.io/

[4] https://github.com/bytecodealliance/wasm-micro-runtime

then executed by a virtual machine. Such a paradigm allows the code to be executed, in theory, on every platform, as long as a VM is available.

WebAssembly manages intra-procedural control flow predictably by using labeled jump-like instructions[5] [7]. This allows for the production of a correct control flow graph of a function by simply analyzing the syntax, whereas in more traditional machine code, such as x86 Assembly, control flow graph recovery is an undecidable problem [8, 9].

Notice also that, in WebAssembly, function calls are performed by either the `call` operator or the `call_indirect` operator: the latter can be compared to calling a function pointer and, as we will see below, is the only source of lack of precision in call graph construction. In WebAssembly function “names" actually are indices, as such, both of call instructions work on numerical indices, either passed directly to the instruction as in `call`, or via the stack in `call_indirect` and referring to an index in a table.

2.2. Call graphs

Formally, the call graph of a program is a directed graph $G = (V, E)$ in which the set of nodes V is the set of functions in the program, and there exists an edge $(f, g) \in E \subseteq V \times V$ if function f calls function g in the program. Apart from the verification of properties such as call ordering constraints, the availability of a call graph is a requirement in many other inter-procedural analyses that need to track the flow of data in a program.

The construction of an *exact* call graph is an undecidable problem, and call graphs are usually computed by over-approximating the possible calls (i.e., the set E contains edges containing all the actual calls, and additional spurious edges). Dynamic features such as function pointers in C/C++ further complicate the issues of computing a call graph. WebAssembly implements function pointers using tables containing references to functions.

In object-oriented programming languages, the main approach to static call graph construction is Class Hierarchy Analysis [10] and its improvements. The main idea of Class Hierarchy Analysis is to employ the inheritance hierarchy to refine the set of methods that could be invoked at a specific program location. WebAssembly does not have a class hierarchy, so this method is not directly applicable. Instead, tools like *Wassail* [4] and *Wasmati* [11] over-approximate calls by looking at the type of the function appearing in a `call_indirect` instruction (i.e., the content of the stack is ignored). A more granular call graph can only be constructed by tracking the values that are on top of the stack when `call_indirect` is used.

2.3. Threats to soundness

WebAssembly modules are designed with host-module interoperability in mind. One of the possible interactions they can have includes the host performing mutations of the running instance of the WebAssembly module. To be more specific, instances of shared memories and tables can be mutated, entailing that with no information available to the analysis about what the host code does, the analysis result may be unsound. For example, our property of interest is the call graph. If the target of a `call_indirect` instruction depends on a value from a memory block mutated by the host code, the analyzer cannot infer this behavior. As another example,

[5] https://webassembly.github.io/spec/core/syntax/instructions.html#syntax-instr-control

we could consider a table containing function references modified by the host at some point during the execution. For the sake of this work, we assume that the module under examination is *closed* and address this issue in Section 6.

2.4. Abstract interpretation

Abstract interpretation [12] is a theory of sound approximation of the semantics of computer programs. It provides a methodology for defining rigorously sound static analyses. Its goal is to derive information about a program's semantics without performing an exhausting exploration of the program under analysis. We obtain this by making the semantics decidable at the cost of precision (Rice's theorem, halting problem) but in a way that the semantics remain sound compared to the concrete semantics. If a concrete interpretation is the evaluation of concrete semantics, an abstract interpretation is the evaluation of abstract semantics. How precise abstract semantics are when compared to concrete semantics is a matter of compromise between computability and tractability of the problem. Due to this factor, tailoring the abstract semantics to the program properties of interest can be an optimal design choice. Abstract interpretation sees application in proving software properties, such as the absence of critical errors, including detecting division-by-zero, NULL pointer dereferencing, etc.

Abstract interpretation makes wide use of upper bound operators, denoted by $\widehat{\sqcup}$ and used to compute control-flow joins, and widening, denoted by $\widehat{\nabla}$ and used in the computation of loops and recursion in a way that ensures termination. These operators are sound, and they influence the precision of the result of the analysis.

For WebAssembly, to-date the only known abstract interpreter capable of producing a call graph is *Sturdy* [5, 13]. It uses only non-relational abstractions, which can affect the precision of the analysis. *Wassail* [4], too, uses notions of abstract interpretation to perform taint analysis.

3. *wassilly*

wassilly (*WebAssembly Ain't So Silly*) is an abstract interpreter that we are actively developing. It is tailored to constructing WebAssembly call graphs but can be extended to verify more general properties, such as the absence of runtime errors.

We introduce and briefly explain the architecture of the tool via the semantics we construct and some of the peculiarities we design for this implementation. We omit the abstract semantics of particular instructions if they are homomorphic to their concrete counterparts.

3.1. Semantics

We introduce concrete and abstract denotational semantics for a simplified version of WebAssembly. Semantics is the rigorous definition of the meaning of the syntactic constructs of a programming language.

The form of semantics we use employs an explicit passing of the next function to compute and is known as continuation semantics. We denote continuations with the symbol χ.

In the context of our semantics, we define a memory σ, holding: a store se comprising of bindings of values to global variable declarations; an indexed table referencing functions; a reference to the topmost element (in-context) in the call (or context) stack, κ. The global store and in-context value stack and local variables are available for reading and updating via specific accessors embedded in σ, respectively: $.se$, $.sk$, $.loc$.

We use an environment ϵ to keep track of local and global variable declarations.

The symbol ϕ is used as a timestamp to differentiate running function instances. ρ is a map from label statements to a tuple consisting of a continuation χ and a transformation τ happening in case branching instructions such as `br` or `br_if` are interpreted with any given label l as the argument.

Our semantics take as inputs: ρ, χ, σ, ϵ, ϕ, κ and produce a function consisting of the continuation χ taking as input the new values for ρ, σ, ϵ, ϕ, κ. The "normal" continuation χ might be overridden with χ' in case of non-linear control flow.

The semantics we defined cover what we deem as the main language concerns, keeping aside `memory` operations, which behave similarly to global variables, hence why they are excluded from this work. Typing is assumed always to be `i32`, 32-bit integer.

Auxiliary functions and notations In order to ease the reading of the semantics we define a set of auxiliary functions. These include basic operations on the stack, boolean evaluation, variable allocation, and a timestamp to differentiate function instances. Some functions that will appear in the semantics will be self-explanatory (e.g: $filter$, *bind*) and are not included in this table.

$let\ pop\ \alpha\ =\ (\alpha', e)$	Pop an element (value, label) from the stack.
$let\ push\ \alpha\ \tilde{\delta}\ =\ \alpha'$	Push an element (value, label) to the stack.
$let\ pop\_twice\ \sigma\ =$ $\quad let\ \sigma',\ v_{right}\ =\ pop\ \sigma\ in$ $\quad let\ \sigma'',\ v_{left}\ =\ pop\ \sigma'\ in$ $\quad (\sigma'', v_{left}, v_{right})$	*pop* is performed twice.
$let\ pop\_n\ \sigma\ n\ = (\sigma', \tilde{v})$	*pop* is performed n times.
$let\ intbool\ \sigma =$ $\quad let\ \sigma', v = pop\ \sigma\ in$ $\quad v = 0\ ?\ (\sigma', false)\ :\ (\sigma', true)$	Concrete int-to-boolean evaluation: 0 is $false$, any non-zero value is $true$.
$let\ \widehat{intbool}\ \widehat{\sigma} =$ $\quad let\ \widehat{\sigma}', \widehat{e} = pop\ \widehat{\sigma}\ in\ \widehat{\sigma}', \widehat{e}$	Abstract int-to-boolean evaluation: the expression $\widehat{e}$ is evaluated later.
$let\ tick\ \phi\ =\ \phi'$	Timestamp generator.
$let\ local\ i\ =\ local\ location$	Allocates a local variable.
$let\ global\ i\ =\ global\ location$	Allocates a global variable.

Table 1: Auxiliary functions, briefly explained.

3.1.1. Concrete semantics

In Table 2 we define the concrete semantics of the reduced version of the WebAssembly instruction set we studied. To avoid ambiguities, we provide a brief description of the semantics.

Concrete semantics	Brief description
$\llbracket c_1; c_2 \rrbracket_{\rho\chi\sigma\epsilon\phi\kappa}\ =\ \llbracket c_1 \rrbracket_{\rho}\ (\llbracket c_2 \rrbracket_{\chi})_{\sigma\epsilon\phi\kappa}$	Sequence of commands
$\llbracket unop \rrbracket_{\rho\chi\sigma\epsilon\phi\kappa}\ =$ $let\ \sigma',\ v\ =\ pop\ \sigma.sk\ in$ $\chi(push\ \sigma'\ [unop\ v])\rho\epsilon\phi\kappa$	Class of operations consisting in: pop once, perform a computation, push the result to the stack.
$\llbracket binop \rrbracket_{\rho\chi\sigma\epsilon\phi\kappa}\ =$ $let\ \sigma',\ v_{left},\ v_{right}\ =\ pop\_twice\ \sigma.sk\ in$ $\chi(push\ \sigma'\ [binop\ v_{left}\ v_{right}])\rho\epsilon\phi\kappa$	Class of operations consisting in: pop twice, perform a computation, push the result to the stack.
$\llbracket local\ i\ t \rrbracket_{\rho\chi\sigma\epsilon\phi\kappa}\ =$ $let\ \epsilon', newvar\ =\ new\epsilon\ \epsilon\ \phi\ (local\ i)\ in$ $let\ \sigma'\ =\ bind\ \sigma\ \phi\ 0\ newvar\ in$ $\chi\sigma'\rho\epsilon'\phi\kappa$	Declaration of local variable (typing t is ignored)
$\llbracket local.get\ i \rrbracket_{\rho\chi\sigma\epsilon\phi\kappa}\ =$ $\chi(push\ \sigma.sk\ [\sigma.loc(i)])\rho\epsilon\phi\kappa$	Read the value of a local variable, then push it to the stack
$\llbracket local.set\ i \rrbracket_{\rho\chi\sigma\epsilon\phi\kappa}\ =$ $let\ \sigma', v\ =\ pop\ \sigma.sk\ in$ $\chi(\sigma'.loc[i \leftarrow v])\rho\epsilon\phi\kappa$	Update a local variable
$\llbracket global\ i\ t \rrbracket_{\rho\chi\sigma\epsilon\phi\kappa}\ =$ $let\ \epsilon', newvar\ =\ new\epsilon\ \epsilon\ \phi\ (global\ i)\ in$ $let\ \sigma'\ =\ bind\ \sigma\ \phi\ 0\ newvar\ in$ $\chi\sigma'\rho\epsilon'\phi\kappa$	Declaration of global variable (typing t is ignored)
$\llbracket global.get\ i \rrbracket_{\rho\chi\sigma\epsilon\phi\kappa}\ =$ $\chi(push\ \sigma.sk\ [\sigma.se.glob(i)])\rho\epsilon\phi\kappa$	Read the value of a global variable, then push it to the stack.
$\llbracket global.set\ i \rrbracket_{\rho\chi\sigma\epsilon\phi\kappa}\ =$ $let\ \sigma', v\ =\ pop\ \sigma.sk\ in$ $\chi(\sigma'.se.glob[i \leftarrow v])\rho\epsilon\phi\kappa$	Update a global variable.
$\llbracket label\ l\ (c) \rrbracket_{\rho\chi\sigma\epsilon\phi\kappa}\ =$ $let\ \sigma', \tilde{v}\ =\ pop\_n\ \sigma.sk\ (arity\ l)\ in$ $let\ \sigma''\ =\ push\ (push\ \sigma'.sk\ [l])\ \tilde{v}\ in$ $fix(\lambda\Theta.\llbracket c \rrbracket_{\rho[l \leftarrow \tau, \Theta]\ \chi\sigma'''\epsilon\phi\kappa})$	Evaluation of a labeled block, it can be either Block or Loop. A fix operator is used to ensure termination in Loops.
$\llbracket br\ l \rrbracket_{\rho\chi\sigma\epsilon\phi\kappa}\ = let\ \tau, \chi'\ = \rho\ l\ in$ $\chi'(\tau\sigma')\rho\epsilon\phi\kappa$	Unconditional branching.
$\llbracket br\_if\ l \rrbracket_{\rho\chi\sigma\epsilon\phi\kappa}\ = let\ \tau, \chi'\ = \rho\ l\ in$ $let\ \sigma', t\ =\ intbool\ \sigma\ in$ $t\ ?\ \chi'(\tau\sigma')\rho\epsilon\phi\kappa\ :\ \chi\sigma'\rho\epsilon\phi\kappa$	Conditional branching.
$\llbracket if\ then\ b_t\ (c_t)\ else\ b_f\ (c_f) \rrbracket_{\rho\chi\sigma\epsilon\phi\kappa} =$ $let\ \sigma', t\ =\ intbool\ \sigma\ in$ $t\ ?\ \llbracket label\ b_t\ (c_t) \rrbracket_{\rho\chi\sigma'\epsilon\phi\kappa}\ : \llbracket label\ b_f\ (c_f) \rrbracket_{\rho\chi\sigma'\epsilon\phi\kappa}$	Evaluation of if-then-else. b_t, b_f are Block labels.

Concrete semantics	Brief description
$[\![fun\ (\tilde{p})\ (\tilde{r})\ (c)]\!]_{\rho\chi\sigma\epsilon\phi\kappa} =$ $let\ \phi' \ =\ tick\ \phi\ in$ $let\ \epsilon', newvars\ =\ new\epsilon\ \epsilon\ \phi'\ (decl\ \tilde{p})\ in$ $let\ \sigma'\ =\ bind\ \sigma\ \phi'\ (val\ \tilde{p})\ newvars\ in$ $let\ \tau\ =\ fun\tau\ \tilde{r}\ in$ $let\ \kappa'\ =\ push\ \kappa\ (\chi, \epsilon, \tau)\ in$ $fix(\lambda\Theta.[\![c]\!]_{\rho\chi(\Theta\to\sigma')\epsilon'\phi'\kappa'})$	Evaluation of a function definition: as a definition can be recursive, we use the fix operator.
$[\![call\ f_i]\!]_{\rho\chi\sigma\epsilon\phi\kappa} =$ $let\ f_{body}\ =\ body\ M.funcs[i]\ in$ $let\ \sigma', \tilde{v}\ =\ pop\_n\ \sigma.sk\ (arity\ M.funcs[i].t)\ in$ $let\ \widetilde{params}, \tilde{r}\ = mod.funcs[i].params,$ $mod.funcs[i].return\ in$ $let\ \tilde{p}\ =\ map\ (\lambda fp, v.(fp, v))\ \widetilde{params}, \tilde{v}\ in$ $[\![fun\ (\tilde{p})\ (\tilde{r})\ (f_{body})]\!]_{\rho\chi\sigma'\epsilon\phi\kappa}$	Evaluation of a direct function call: the body is retrieved from M, the function parameters are bound, then the body is evaluated.
$[\![call\_indirect\ t_i]\!]_{\rho\chi\sigma\epsilon\phi\kappa} =$ $let\ \sigma', f_i\ =\ pop\ \sigma\ in$ $let\ f_{idx}\ i\ =\ \sigma.se.funtable[i]\ in$ $[\![call\ (f_{idx}\ i)]\!]_{\rho\chi\sigma'\epsilon\phi\kappa}$	Evaluation of function indirection. The reference is held on a table, and it is retrieved by the function $f_{idx}\ i$.
$[\![return]\!]_{\rho\chi\sigma\epsilon\phi\kappa} =$ $let\ \kappa', \chi', \epsilon', \tau =\ pop\ \kappa\ in$ $\chi'(\tau\sigma)\rho\epsilon\phi\kappa'$	Evaluation of return from function.

Table 2: Concrete semantics for the subset of the WebAssembly language we studied.

3.1.2. Abstract semantics

In Table 3 we introduce the abstract semantics functions. As already mentioned, we only introduce the semantics of instructions that are non-homomorphic to their concrete counterparts. The descriptions on the side provide an explanation of how the abstract semantic functions differ from their concrete version.

Abstract semantics	Brief description
$[\![br\ l]\!]_{\rho\chi\widehat{\sigma}\widehat{\epsilon}\phi\widehat{\kappa}}\ =$ $let\ \tau, \chi'\ = \rho\ l\ in$ $let\ \widehat{\sigma}'\ =\ \tau\widehat{\sigma}\ in$ $let\ \rho', \chi''\ =\ case\ l\ of\ :$ $loop \to\ let\ r\ = \chi'\ \widehat{\nabla}\ \widehat{\sigma}'\ in$ $\rho[l \leftarrow (r), \tau], r$ $block \to\ let\ r\ =\ \chi'\ \widehat{\sqcup}\ \widehat{\sigma}'\ in$ $\rho[l \leftarrow (r), \tau], r$ $in\ \chi''\widehat{\sigma}'\rho'\widehat{\epsilon}\phi\widehat{\kappa}$	Abstract br, with respect to its concrete counterpart, computes the widening ($\widehat{\nabla}$) or the upper bound ($\widehat{\sqcup}$) between the “old" computation χ', and the “new" one, $\widehat{\sigma}'$. With this mechanism we obtain χ''. The $\widehat{\nabla}$ operator ensures termination.

Abstract semantics	Brief description
$[\![br\_if\ l]\!]_{\rho\chi\widehat{\sigma}\widehat{\epsilon}\phi\widehat{\kappa}} =$ $\quad let\ \tau, \chi' = \rho\ l\ in$ $\quad let\ \widehat{\sigma}', \widehat{e} = \widehat{intbool}\ \widehat{\sigma}\ in$ $\quad let\ \widehat{\sigma}'_t = \tau(filter\ \widehat{e}\ \widehat{\sigma}')\ in$ $\quad let\ \widehat{\sigma}'_f = filter\ \neg\widehat{e}\ \widehat{\sigma}'\ in$ $\quad let\ \rho', \chi'' = case\ l\ of:$ $\qquad loop \rightarrow let\ r = \chi'\ \widehat{\nabla}\ \widehat{\sigma}'_t\ in\ \rho\,[l \leftarrow (r), \tau], r$ $\qquad block \rightarrow let\ r = \chi'\ \widehat{\sqcup}\ \widehat{\sigma}'_t\ in\ \rho\,[l \leftarrow (r), \tau], r$ $\quad in\ \chi''\widehat{\sigma}'_t\rho'\widehat{\epsilon}\phi\widehat{\kappa}\ \widehat{\sqcup}\ \chi\widehat{\sigma}'_f\rho\widehat{\epsilon}\phi\widehat{\kappa}$	A function $filter$ is applied. This improves the precision of the analysis. $\widehat{\sigma}'_t$, $\widehat{\sigma}'_f$ are the memories in which, respectively, $\widehat{e}$ is true and false.
$[\![if\ then\ b_t\ (c_t; end)\ else\ b_f\ (c_f; end)]\!]_{\rho\chi\widehat{\sigma}\widehat{\epsilon}\phi\widehat{\kappa}} =$ $\quad let\ \widehat{\sigma}', \widehat{e} = \widehat{intbool}\ \widehat{\sigma}\ in$ $\quad [\![label\ b_t\ (c_t; end)]\!]_{\rho\chi(filter\ \widehat{e}\ \widehat{\sigma}')\widehat{\epsilon}\phi\widehat{\kappa}}\ \widehat{\sqcup}$ $\qquad [\![label\ b_f\ (c_f; end)]\!]_{\rho\chi(filter\ \neg\widehat{e}\ \widehat{\sigma}')\widehat{\epsilon}\phi\widehat{\kappa}}$	Computes the upper bound between the computations in b_t and b_f. Similarly to *br_if*, $filter$ is applied.
$[\![call\_indirect\ t_i]\!]_{\rho\chi\widehat{\sigma}\widehat{\epsilon}\phi\widehat{\kappa}} =$ $\quad let\ \widehat{\sigma}', \widehat{f}_i = pop\ \widehat{\sigma}\ in$ $\quad let\ f_{idx}\ i = \widehat{\sigma}.se.funtable[i]\ in$ $\quad \widehat{\sqcup}(\{\forall\ i \in \widehat{\gamma}(\widehat{f}_i)\} \mid [\![call\ (f_{idx}\ i)]\!]_{\rho\chi\widehat{\sigma}'\widehat{\epsilon}\phi\widehat{\kappa}}\})$	Computes the upper bound between the calls to the possible callees. Target functions are type filtered. The notation omits this detail for triviality and readability.

Table 3: Abstract semantics for the subset of the WebAssembly language we studied.

In this semantic notation, we omit the specific semantics of the call graph for readability. Briefly explained: any instruction that is not `call i` or `call_indirect t` will produce an empty set of callees ϕ. The `call i` instruction will produce a singleton `{i}`, and the `call_indirect` instruction will produce a set containing the function indexes obtained by resolving the indirections generated by concretizing the abstract value on top of the stack. As noted, we also filter possible callee functions that do not match the type `t`.

3.2. Architecture of *wassilly*

As mentioned, *wassilly* is a specialized tool for constructing call graphs in a “reasonably" precise, yet sound, manner. We can achieve this by deducing what value(s) can be on top of the stack when a `call_indirect` instruction is met. This can be done with a varying degree of precision. For our work, we use a relational abstraction to represent values. We then build specialized data structures on top of it to represent the operand stack, the call stack, and the linear memory. As `memory` operations are outside the scope of this work, and the call stack does not present any particular novelty, we will not discuss those and will focus solely on the operand stack.

One of the peculiarities of our implementation is its capability to analyze WebAssembly code directly as stack machines, relationally, and lazily. This is not the case for other analyzers for languages based on stack machines. For example, *Sturdy* [13, 5], to the best of our knowledge, is only capable of non-relational analyses when considering stack machines. Other tools, such as *Soot* [14], *Wimpl* [15], and *cmm_of_wasm*[6] employ some kind of explicit language transformation step.

The second peculiarity of our implementation of the operand stack consists of lazily building abstract expressions on the fly, without necessarily evaluating them (i.e. if we are adding two integer operands together and we are not interested in integer overflows/underflows, we do not evaluate the expression, we pop `e1`, `e2` from the stack and push the resulting mathematical expression `e1+e2` to the operand stack). This applies only to the cases in which it is possible to construct an expression representing the content of the stack lazily. For instance, bitwise operations are not performed this way.

3.3. Discussion

As the tool is under development a real experimental evaluation cannot be performed yet. Ignoring performance issues, we demonstrate how the tool operates on the common language concerns in WebAssembly. The systematic literature review in [3] exhaustively showcases all of the language concerns when we are interested in building call graphs.

We will also make some brief considerations about the language concerns outside the scope of this paper: mutations performed by the host and `memory` instructions.

The work presented in [3] assesses that the main language concerns that are computed in an unsound manner are the reachability of functions in imported and exported tables. In WebAssembly the reachability of references in exported tables should be given for granted, as they are equivalently reachable as exported functions. It can be helpful to think of them as a mutable set of exported function references as opposed to a set of exported functions.

Regarding imported functions, while we do not have any information about the behavior of such functions, unless some form of extra information is available, we can always asses that they are reachable and can give an unconstrained function output of the correct typing, as it is explicit in the function signature.

Besides these points and concerns relative to unknown imported objects, we are potentially able to cover most of the rest of the concerns with at least the same degree of precision as the state of the art, while retaining soundness, as we use inference rules which consider the semantics of a given program instead of type inference only, and a relational abstraction that allows to derive stronger properties.

Some specific language concerns, such as “10: Table init. offset is imported from host" would need some form of specification to provide a conclusive analysis. This means that with no additional information available we either produce a sound and very imprecise analysis or an unsound one, which is not a desirable result for the scope of this work. This need for additional information takes us to the next point: table mutability.

The concern of table mutability performed by the host is, in a fundamental way, similar to the previous one: if we are to produce a sound analysis we should assume that calling a host function can mutate the tables, and without information available we are “forced" to be conservative and assume we do not know what function reference is held at any given table entry, making the analysis of some programs as imprecise as an analysis based on function types only.

[6] https://github.com/SimonJF/cmm_of_wasm

We can generalize these concerns as “Target callee is dependant on host behavior". This covers the concerns: “9: Table is mutated by host", “10: Table init. offset is imported from host", “11: Memory init. offset is imported from host".

Further indications about this issue will be discussed in Section 6.

4. Related work

WebAssembly was first released in 2017 [16] and can therefore be considered a relatively “young" language. The formal specification and a reference implementation in OCaml are available online [17]. An Isabelle specification is also available in [18].

The work presented in [19] shows how WebAssembly can be used as an attack vehicle to write to memory and cause unexpected behavior in the host environment. The tool *Wasmati* introduced in [11] employs a graph-based representation of WebAssembly code, called Code Property Graphs (CPG), and checks security properties on these using a query language. Wasmati requires a call graph but can generate one by considering function signatures only.

Even in the absence of buffer overflows and similar attacks, it is still possible for WebAssembly to leak information in unintended ways; to address this issue, recent work has focused on information flow analysis, see for instance [7] and [20].

Tools that focus, more in general, on static analysis of WebAssembly include *WASP* [21], *Manticore* [6], and *Wassail* [4]. The first two tools are based, respectively, on symbolic and concolic execution. Wassail, instead, can be seen as a library to develop analyses. *Wassail* also supports *program slicing* [22] and the computation of control flow graphs and call graphs. In particular, similar to *Wasmati*, it can provide an over-approximation of the call graph for a specific WebAssembly module by relying on the types of functions invoked in indirect calls (see Section 2). *Sturdy* [5, 13] presents another framework to develop and perform static analysis. It is a modular, extensible framework based on abstract interpretation. It supports WebAssembly bytecode and out-of-the-box can perform analyses such as dead code analysis and taint analysis.

The work presented in [15] uses a different approach to verifying WebAssembly code. The authors introduce a simplified C-like Intermediate Representation called *Wimpl*. The goal is to use existing tools for C instead of developing verification tools from scratch.

The work in [3] presents a comparison between tools that are capable of constructing a call graph for WebAssembly dynamically or statically, introducing a set of micro-benchmarks that allow the assessment of the precision of a certain analyzer via specifically crafted edge cases. Such edge cases cover the language concerns regarding indirect calls.

5. Conclusion

Given the increasing importance of WebAssembly and the importance of having static analysis support, we have given concrete and abstract semantics for a simplified version of the WebAssembly language. We also highlighted some additional challenges which we do not cover in this work and propose means to tackle them.

The abstract semantics we specified form the basis of an analyzer based on the theory of Abstract Interpretation that is currently being developed.

We also described the approach we are pursuing in the implementation of the analyzer, which features a lazy operand stack and the ability to directly analyze WebAssembly bytecode relationally. Based on this, we can predict our approach to be at least as precise as the sound ones present in the state-of-the-art, whilst potentially being capable of tackling language concerns which, at the moment, are being computed in an unsound manner.

6. Future work

The ability of WebAssembly to work as an external mechanism to provide high-performance computations is what makes it so popular. Unfortunately, we cannot assume in a definitive manner that all WebAssembly modules work in isolation and are closed. Some possible future work in improving the analysis of WebAssembly modules can and should include some mechanism for defining or inferring host-module(s) interactions.

This includes analysis of imported host functions (and subsequently analysis of the functions depending on them) and production of function summaries or a form of specification in a custom Domain Specific Language.

Trivially, a DSL would allow to manually set the rules we would instead infer, and would work as a sort of specification contract (i.e: the soundness of the analysis is guaranteed with respect to the contract). A similar approach has been explored in [23] for symbolic execution but, to the best of our knowledge, no approaches are available for abstract interpretation.

Both of these approaches have pros and cons: the former keeps the analysis fully automated, but the analysis still has to maintain soundness at the cost of precision. The latter requires the handwriting of a specification and a degree of knowledge about the specific host program interacting with the WebAssembly module: it is potentially more precise, as false positives can be ruled out, but it is prone to be influenced by human error.

References

[1] A. Ene, How prime video updates its app for more than 8,000 device types, https://www.amazon.science/blog/how-prime-video-updates-its-app-for-more-than-8-000-device-types, 2022. Accessed: 2022-10-11.

[2] T. Schroeder, D. Jordan, S. Schulz, R. Cain, M. Hanley, M. Fay, Introducing the disney+ application development kit (adk), https://medium.com/disney-streaming/introducing-the-disney-application-development-kit-adk-ad85ca139073, 2021. Accessed: 2022-10-10.

[3] D. Lehmann, M. Thalakottur, F. Tip, M. Pradel, That's a tough call: Studying the challenges of call graph construction for webassembly, in: R. Just, G. Fraser (Eds.), Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023, ACM, 2023, pp. 892–903. URL: https://doi.org/10.1145/3597926.3598104. doi:10.1145/3597926.3598104.

[4] Q. Stiévenart, C. De Roover, Wassail: a webassembly static analysis library, in: Fifth International Workshop on Programming Technology for the Future Web, 2021.

[5] K. Brandl, S. Erdweg, S. Keidel, N. Hansen, Modular abstract definitional interpreters for webassembly, in: K. Ali, G. Salvaneschi (Eds.), 37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States, volume 263 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, pp. 5:1–5:28. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2023.5. doi:10.4230/LIPIcs.ECOOP.2023.5.

[6] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, A. Dinaburg, Manticore: A user-friendly symbolic execution framework for binaries and smart contracts, in: 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019, IEEE, 2019, pp. 1186–1189. URL: https://doi.org/10.1109/ASE.2019.00133. doi:10.1109/ASE.2019.00133.

[7] Q. Stiévenart, C. De Roover, Compositional information flow analysis for webassembly programs, in: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, 2020, pp. 13–24.

[8] M. H. Nguyen, T. B. Nguyen, T. T. Quan, A hybrid aproach for control flow graph construction from binary code, in: 20th Asia-Pacific Software Engineering Conference, 2013.

[9] A. Flores-Montoya, E. M. Schulte, Datalog disassembly, in: S. Capkun, F. Roesner (Eds.), 29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020, USENIX Association, 2020, pp. 1075–1092. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/flores-montoya.

[10] J. Dean, D. Grove, C. Chambers, Optimization of object-oriented programs using static class hierarchy analysis, in: European Conference on Object-Oriented Programming, Springer, 1995, pp. 77–101.

[11] T. Brito, P. Lopes, N. Santos, J. F. Santos, Wasmati: An efficient static vulnerability scanner for webassembly, Computers & Security 118 (2022) 102745. URL: https://www.sciencedirect.com/science/article/pii/S0167404822001407. doi:https://doi.org/10.1016/j.cose.2022.102745.

[12] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: R. M. Graham, M. A. Harrison, R. Sethi (Eds.), Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977, ACM, 1977, pp. 238–252. URL: https://doi.org/10.1145/512950.512973. doi:10.1145/512950.512973.

[13] S. Keidel, S. Erdweg, T. Hombücher, Combinator-based fixpoint algorithms for big-step abstract interpreters, Proc. ACM Program. Lang. 7 (2023) 955–981. URL: https://doi.org/10.1145/3607863. doi:10.1145/3607863.

[14] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, V. Sundaresan, Soot - a java bytecode optimization framework, in: S. A. MacKay, J. H. Johnson (Eds.), Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada, IBM, 1999, p. 13. URL: https://dl.acm.org/citation.cfm?id=782008.

[15] M. Thalakottur, F. Tip, D. Lehmann, M. Pradel, Wimpl: A simple ir for static analysis of webassembly binaries, in: Program Analysis for WebAssembly (PAW) 2022, 2022.

[16] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai,

J. 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, pp. 185–200.

[17] A. Rossberg, WebAssembly Core Specification, 2019. URL: https://www.w3.org/TR/wasm-core-1/.

[18] C. Watt, Mechanising and verifying the webassembly specification, in: Proceedings of the 7th ACM SIGPLAN International Conference on certified programs and proofs, 2018, pp. 53–65.

[19] D. Lehmann, J. Kinder, M. Pradel, Everything old is new again: Binary security of WebAssembly, in: 29th USENIX Security Symposium (USENIX Security 20), USENIX Association, 2020, pp. 217–234. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann.

[20] I. Bastys, M. Algehed, A. Sjösten, A. Sabelfeld, Secwasm: Information flow control for webassembly, in: Program Analysis for WebAssembly (PAW), 2022.

[21] F. Marques, J. Fragoso Santos, N. Santos, P. Adão, Concolic Execution for WebAssembly, in: K. Ali, J. Vitek (Eds.), 36th European Conference on Object-Oriented Programming (ECOOP 2022), volume 222 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2022, pp. 11:1–11:29. URL: https://drops.dagstuhl.de/opus/volltexte/2022/16239. doi:10.4230/LIPIcs.ECOOP.2022.11.

[22] Q. Stiévenart, D. W. Binkley, C. De Roover, Static stack-preserving intra-procedural slicing of webassembly binaries, in: Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 2031–2042.

[23] N. He, Z. Zhao, J. Wang, Y. Hu, S. Guo, H. Wang, G. Liang, D. Li, X. Chen, Y. Guo, Eunomia: Enabling user-specified fine-grained search in symbolically executing webassembly binaries, in: R. Just, G. Fraser (Eds.), Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023, ACM, 2023, pp. 385–397. URL: https://doi.org/10.1145/3597926.3598064. doi:10.1145/3597926.3598064.

Symbols

M	WebAssembly module
P	Program (instruction, function)
ϵ	Environment
$\widehat{\epsilon}$	Abstract environment
$\widehat{\sigma}$	Abstract memory
σ	Memory
χ	Continuation
τ	Transformation
$\mathcal{I}$	Interpreter
l	Label
ρ	Map $l \rightarrow (\tau, \chi)$
ϕ	Timestamp
κ	Context
$\widehat{\kappa}$	Abstract context
$\widehat{\gamma}$	Concretization
$\widehat{\sqcup}$	Union
$\widehat{\nabla}$	Widening