Concolic Execution for WebAssembly

Filipe Marques
Instituto Superior Técnico, University of Lisbon, Portugal
INESC-ID Lisbon, Portugal

José Fragoso Santos
Instituto Superior Técnico, University of Lisbon, Portugal
INESC-ID Lisbon, Portugal

Nuno Santos
Instituto Superior Técnico, University of Lisbon, Portugal
INESC-ID Lisbon, Portugal

Pedro Adão
Instituto Superior Técnico, University of Lisbon, Portugal
Instituto de Telecomunicações, Aveiro, Portugal

Abstract

WebAssembly (Wasm) is a new binary instruction format that allows targeted compiled code written in high-level languages to be executed by the browser's JavaScript engine with near-native speed. Despite its clear performance advantages, Wasm opens up the opportunity for bugs or security vulnerabilities to be introduced into Web programs, as pre-existing issues in programs written in unsafe languages can be transferred down to cross-compiled binaries. The source code of such binaries is frequently unavailable for static analysis, creating the demand for tools that can directly tackle Wasm code. Despite this potentially security-critical situation, there is still a noticeable lack of tool support for analysing Wasm binaries. We present WASP, a symbolic execution engine for testing Wasm modules, which works directly on Wasm code and was built on top of a standard-compliant Wasm reference implementation. WASP was thoroughly evaluated: it was used to symbolically test a generic data-structure library for C and the Amazon Encryption SDK for C, demonstrating that it can find bugs and generate high-coverage testing inputs for real-world C applications; and was further tested against the Test-Comp benchmark, obtaining results comparable to well-established symbolic execution and testing tools for C.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging; Security and privacy → Formal methods and theory of security

Keywords and phrases Concolic Testing, WebAssembly, Test-Generation, Testing C Programs

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.11

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.20`

Funding The authors were supported by national funds through Fundação para a Ciência e a Tecnologia (UIDB/50008/2020, Instituto de Telecomunicações, and UIDB/50021/2020, INESC-ID multi-annual funding), projects INFOCOS (PTDC/CCI-COM/32378/2017) and DIVINA (CMU/T-IC/0053/2021), and by the European Commission under grant agreement number 830892 (SPARTA).

Acknowledgements We are grateful to Carolina Costa with whom we designed a preliminary version of WASP and its concolic semantics as part of her M.Sc. thesis [22].

1 Introduction

WebAssembly (Wasm) [30] is a binary instruction format designed to be the new standard compilation target for the Web and is now supported by all major browser vendors, enabling Web applications to run with near-native speed. As a result, Web applications are increasingly being ported into Wasm to reap its performance benefits. In particular, Wasm has been adopted in server-side runtimes [13, 1, 20], IoT platforms [31], and in edge computing [34].

The compilation of unsafe languages to Wasm opens up the opportunity for the introduction of new classes of bugs and security vulnerabilities into the setting of Web programs, as such issues in the original programs can be transposed to Wasm binaries via compilation [23]. This is the case, for example, of buffer overflows [50], format string bugs [9], and use-after-free errors [29]. By exploiting such flaws [25], cyber attackers have access to a widened surface for launching their attacks on the Web. These include cross-site scripting attacks by exploiting client-side code [50], or code injection attacks by targeting vulnerabilities in server-side code (e.g., powered by Node.js). In addition, Wasm itself can be used for writing malware, e.g., web keyloggers [49], or crypto-miners [59]. Importantly, Wasm binaries are often integrated directly into Web applications, with developers not having access to the corresponding source code. In this scenario, developers must analyse stand-alone Wasm code to test it against potential security vulnerabilities and other types of execution errors.

Symbolic execution [15, 5] is a program analysis technique that allows for the exploration of multiple program paths by running the given program using symbolic values instead of concrete ones. It has successfully been applied to finding a wide range of security vulnerabilities and other types of bugs in many high-level and intermediate languages, including C [14, 28], Java [54], and JavaScript [61, 60, 26]. Nonetheless, to the best of our knowledge, there are only two tools for symbolically executing Wasm code: WANA [73] and Manticore [51]. Both these tools are, however, mainly targeted at the analysis of smart contracts and have important limitations that constraint their application to stand-alone Wasm modules. WANA [73] is in preliminary development stages and can only be applied to EOSIO and Ethereum smart contracts, since it does not include a symbolic execution engine for Wasm that can be run on its own. Manticore [51] has recently gained support for Wasm [33], but has not yet been systematically evaluated on Wasm code. Furthermore, its application to Wasm modules requires the manual setup of a complex Python script for each possible input memory, making it cumbersome and difficult to automate.

We present the WebAssembly Symbolic Processor, WASP, a novel concolic execution engine for testing Wasm (version 1.0) modules. WASP follows the so-called *concolic discipline* [28, 64], combining concrete execution with symbolic execution and exploring one execution path at a time. However, unlike most concolic execution engines [65, 79, 64, 63], which are implemented via program instrumentation, we implement WASP by instrumenting the Wasm reference interpreter developed by Haas et al. [30]. To this end, we lift the authors' reference interpreter from concrete values to pairs of concrete and symbolic values. By moving the instrumentation to the interpreter level, we open up the possibility for a range of optimisations in the context of concolic execution. In this paper, we explore two such optimisations: **(1)** application of algebraic simplifications to byte-level symbolic expressions generated by memory interactions (§3.3); and **(2)** shortcut restarts for failed assumption statements (§3.4). Finally, we formalise our concolic analysis as a small-step concolic semantics, which we use to both guide the implementation of WASP and describe its mathematical underpinnings. This semantics is an additional contribution of the paper as we are not aware of any such formalisation of concolic execution for a low-level language.

While our first goal is for WASP to be able to analyse stand-alone Wasm modules, we also aim for it to be used as a common platform for building symbolic analyses for high-level programming languages that compile to Wasm. In a nutshell, if one wants to use WASP to enable a symbolic execution engine for a given language, one has to accomplish the following two tasks. First, the symbolic primitives of WASP, such as the declaration of assertions, assumptions, and the creation of symbolic variables, must be exposed at the source-language level and properly connected to the corresponding WASP primitives via compilation. Second,

one must guarantee that either the code of the required runtime libraries is available for symbolic execution or that WASP includes symbolic summaries that model the behaviour of those libraries. In order to demonstrate the viability of this approach, we use it to build WASP-C, a new symbolic execution framework for testing C programs. WASP-C shows that, with a relatively small effort ($\approx$ 800 LOC), we were able to build a new concolic engine for C that is able to analyse industry-grade code and obtain results comparable to well-established symbolic execution and testing tools for C, such as KLEE [14] and VeriFuzz [7].

We evaluate WASP in the five following ways: **(1)** We compare the performance of WASP with that of Manticore [51] in the analysis of a stand-alone Wasm B-tree data structure [22]. We demonstrate that WASP outperforms Manticore, its only competitor tool. **(2)** We use WASP-C to symbolically test Collections-C [52], a widely-used generic data structure library for C previously tested using the Gillian-C tool [26]. WASP-C found three bugs during the testing process, including a previously unknown bug that Gillian-C did not detect. Also, WASP-C is more efficient than Gillian-C, completing the symbolic analysis of the library 14% faster. **(3)** We run WASP-C against the Test-Comp [10] benchmark, obtaining results comparable to well-established symbolic execution and testing tools for C, such as KLEE [14] and VeriFuzz [7]. If we compare the results we obtained for WASP-C against those obtained for the tools submitted for the 2021 Competition on Software Testing (TestComp 2021 [11]), we conclude that WASP-C is the third-best tool in the *cover-error* category and the sixth-best tool in the *cover-branches* category out of a total of twelve tools (with WASP included). **(4)** We measure the impact of the proposed optimisation techniques on the performance of WASP by comparing the execution times obtained for WASP with the optimisations enabled against those obtained with them disabled. Results indicate the proposed optimisations are essential for WASP's performance. **(5)** We use WASP-C to symbolically test the Amazon Encryption SDK [67], generating a high-coverage test suite for that library and demonstrating that WASP-C scales to industry-grade code.

Our evaluation is mostly focused on WASP-C due to the fact that there is no symbolic benchmark for stand-alone Wasm modules. However, according to a recent study [35], most Wasm code on the Web ($\approx$ 65%) comes from C/C++ applications. In this light, we believe it is appropriate to center the analysis of the performance of WASP on compiled C code.

Contributions. In summary, the contributions of this work are three-fold: **(1)** WASP, a concolic execution engine for Wasm (§3); **(2)** WASP-C, a symbolic execution framework for testing C programs built on top of WASP (§3.5); and **(3)** three symbolic datasets for evaluating symbolic execution tools for Wasm, covering different types of symbolic reasoning (§4).

2 Background

In this section we give an overview of Wasm, focusing on its syntax and semantics, together with a high-level introduction to symbolic execution with a particular emphasis on the concolic approach followed in this paper.

2.1 WebAssembly

WebAssembly [30, 57] is a low-level bytecode format that offers compact representation, efficient validation and compilation, and ensures safe execution with minimal overhead. Wasm is not tied up to any specific hardware, being language-, hardware- and platform-independent. Like other assembly languages, it is mainly used as a compilation target for high-level languages, such as C/C++ or Rust, allowing for code written in a range of languages to be run on web browsers with significant speed improvements compared to JavaScript [78].

Value Types
$t ::= i32 \mid i64 \mid f32 \mid f64$

Function Types
$tf ::= t^* \to t^*$

Instructions
$e ::=$ `unreachable` | `nop` | `drop` | `select` | t`.const` | $t.unop_t$ | $t.binop_t$ | $t.testop_t$ | $t.relop_t$ | $t.cvtop\ t\_sx^?$ | {`get`|`set`|`tee`}`_local` i | {`get`|`set`}`_global` i | t`.load (tp_sx`$^?$`)` $a\ o$ | t`.store tp`$^?$ $a\ o$ | `current_memory` | `grow_memory` | `block` $tf\ e^*$ `end` | `loop` $tf\ e^*$ `end` | `if` $tf\ e^*$ `else` e^* `end` | `br` i | `br_if` i | `br_table` i^+ | `return` | `call` i | `call_indirect` tf

Imports
$im ::=$ `import` "$name$" "$name$"

Exports
$ex ::=$ `export` "$name$"

Functions
$f ::= ex^*$ `func` tf `local` $t^*\ e^* \mid ex^*$ `func` $tf\ im$

Figure 1 Simplified Wasm abstract syntax, as presented in [30].

Syntax. A WebAssembly binary takes the form of a *module*. A Wasm module includes a collection of Wasm *functions*, together with the declaration of their shared global variables and the specification of the linear memory where the functions and global variables are loaded. Computation is based on a *stack machine*; Wasm instructions interact with the stack by pushing values onto the stack or popping values out of the stack. A Wasm module is executed by an *embedder*, e.g., the host JavaScript engine that defines how modules are loaded, resolves imports and exports between modules, and handles I/Os, timers, and traps.

The syntax of Wasm programs is given in Figure 1 and includes: functions f, instructions e, values c, value types t, and function types tf. Wasm has four *primitive types*, all readily available on common hardware: machine-integers and IEEE 754 floating-point numbers [36], each with a 32- and 64-bit variant. Wasm makes no distinction between signed and unsigned *integers*; instead, instructions have a *sign extension* to indicate how to interpret the generated integer values. *Wasm variables* can be either *local*, belonging to the execution context of a function, or *global*, belonging to the entire module. Wasm does not have "named" variables; instead, both local and global variables are indexed by integer values. The primary storage of a Wasm module is a large array of bytes, commonly referred to as *linear memory*. The initial memory size is fixed. However, memories can be programmatically grown. In contrast to most stack machines, Wasm has structured control flow constructs, ensuring that humans can easily interpret Wasm code and that no irreducible loops [32] are encountered. Wasm *instructions* can be divided into the following categories:

- *Stack instructions*: the instruction `drop` for popping the value at the top of the stack; the instruction `const` for pushing a value onto the stack; the unary and binary operator instructions for applying the corresponding operators to the value(s) at the top of the stack, replacing that(those) value(s) with the obtained result. Operator instructions include the standard relational, arithmetic, and boolean operators.
- *Variable instructions*: the instructions `set_local` and `set_global` for updating the values of local and global variables; the instructions `get_local` and `get_global` for retrieving the values of local and global variables, placing them at the top of the stack; and the instruction `tee_local` for setting the value of a local variable to the value at the top of the stack without removing that value from the stack.
- *Memory instructions*: the instructions `load` and `store` for loading and storing primitive values from and to memory; the instruction `grow_memory` for increasing the size of the current memory one page at a time – page size is fixed at 64 KiB; and the instruction `current_memory` for obtaining the size of the current memory.
- *Control flow instructions*: the standard control-flow instructions: `loop`, `if` and `block`, `br`, `return`, `call`; and the Wasm-specific control-flow instructions `call_indirect`, used to implement dynamic dispatch at the Wasm level, and `br_tbl`, used to implement the standard `switch` statement.

The reader is referred to [30] for a thorough account of the syntax of Wasm.

2.2 Symbolic Execution

Symbolic execution is a program analysis technique used to explore all feasible paths of a program up to a bound [41]. Instead of running a program using concrete values, symbolic execution engines run the given program with *symbolic* inputs. Every time the symbolic execution engine hits a conditional expression with a symbolic guard, the engine forks the current execution to be able to explore both branches. For each execution path, the symbolic execution engine builds a first order formula, called *path condition*, which accumulates the constraints on the symbolic inputs that direct the execution along that path. In particular, every time a conditional instruction is symbolically executed, the current path condition is extended with its guard in the "then" branch and with the negation of its guard in the "else" branch. Symbolic execution engines rely on an underlying SMT solver to check the feasibility of execution paths and the validity of the assertions supplied by the developer. An execution path is said to be feasible if it can be realised by at least one concrete path and an assertion holds at a given program point if it is implied by the path condition at that point.

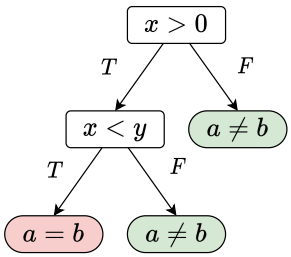
Concolic Execution. Concolic execution is a special variation of symbolic execution in which one pairs up a concrete execution with a symbolic execution to avoid interactions with the underlying SMT solver by exploring one execution path at a time [28, 64]. Concolic execution engines assign concrete values to symbolic inputs and execute the given program both concretely and symbolically at the same time, following only the concrete path but constructing the path condition corresponding to that path as in pure symbolic execution. The constructed path condition is instrumental to concolic execution as it captures the conditions that must hold for the execution to take the explored path. More specifically, it can be used to generate new concrete inputs for symbolic variables that will force the exploration of a different path. To this end, one needs to negate the obtained path condition and query the underlying SMT solver for a model of the obtained formula. By keeping track of all the path conditions generated via concolic execution, the engine can enumerate all program execution paths up to a bound, with the advantage of only having to interact with the underlying constraint solver one time per explored path. Note that in purely symbolic execution, the engine must query the constraint solver every time it hits a branching point in order to determine whether or not its then- and else- branches are feasible.

Concolic Execution: Example. Let us now take a look at how concolic execution works in practice. Consider the C program given in Figure 2a. This program is annotated with a final assert statement, which is supposed to hold independently of the values of variables x and y. Hence, in order to determine whether or not the assertion always holds, one has to explore all feasible execution paths of the program, which we illustrate in Figure 2b in the form of an execution tree. In the tree, we depict in green the leaf nodes corresponding to execution paths for which the assertion holds and in red those corresponding to paths for which it does not. The final assertion does not hold for the left-most path. To see this, consider the inputs $x = 1$ and $y = 4$. These inputs cause variables a and b to be both assigned to 6, violating the final assertion. Below, we explain how these inputs can be discovered.

As there are three possible execution paths, there will be three concolic executions, each corresponding to a different execution path. In the following, we will refer to these executions as *concolic iterations*. During the *first concolic iteration*, the concrete values associated with the symbolic variables of the program are picked non-deterministically from the set of all concrete values of their corresponding types. For this example, we will assume that x and y are respectively set to 0 and 2. These inputs cause the concolic execution engine to explore the rightmost path of the execution tree, generating the final path condition: $x \leq 0$.

```c
int main() {
    int a = 4, b = 2;
    int x = symb(), y = symb();
    if (x > 0) {
        b = a + 2;
        if (x < y)
            a = (x * 2) + y;
    }
    assert(a != b);
}
```

(a) Symbolic C program.



(b) Execution tree for program in Listing 2a.

Figure 2 Concolic execution example.

Before the *second concolic iteration*, the concolic execution engine queries the underlying constraint solver for a model for the symbolic inputs that satisfies the formula $x > 0$, corresponding to the negation of the first path condition. Let us assume that the solver returns the model $x = 1$ and $y = 0$. These inputs cause the concolic execution engine to explore the middle path, generating the path condition: $(x > 0) \wedge (x \geq y)$.

Before the *third concolic iteration*, the engine queries the solver for a model for the symbolic inputs that satisfies the negation of both path conditions found so far:

$$(x > 0) \wedge ((x \leq 0) \vee (x < y)) \;\equiv\; (x > 0) \wedge (x < y)$$

Assume that the solver outputs the model $x = 1$ and $y = 2$. These inputs cause the concolic execution engine to explore the leftmost path of the execution tree. Observe that this model does not immediately trigger the assertion violation, since the final values of a and b do not coincide ($a = 4$ and $b = 6$). In order to understand how the concolic execution engine finds the model that violates the assertion, one has to consider the concolic state at the point where the assert statement is encountered, which is given below:

$$x \mapsto (1, x) \quad y \mapsto (2, y) \quad a \mapsto (4, 2 \times x + y) \quad b \mapsto (6, 6) \quad PC \equiv (x > 0) \wedge (x < y)$$

Given this concolic state, the expression $a \neq b$ evaluates to the concrete value *true* and the symbolic value $(2 \times x + y) \neq 6$. In order to establish that the assertion holds, the concolic execution engine must prove that the *symbolic expression* being asserted is implied by the current path condition; put formally:

$$(x > 0) \wedge (x < y) \Rightarrow (2 \times x + y) \neq 6$$

In order to check the validity of this implication, the concolic execution engine queries the underlying constraint solver for the satisfiability of its negation:

$$(x > 0) \wedge (x < y) \wedge (2 \times x + y) = 6$$

which is satisfied by the model $x = 1$ and $y = 4$, disproving the implication and witnessing the assertion failure.

3 WASP

This section presents our concolic execution engine for Wasm named WASP. We begin by describing the architecture of WASP (§3.1). Next, we explain the concolic semantics (§3.2) and memory model (§3.3) at the core of WASP. Then, we introduce an optimised version of the proposed concolic semantics (§3.4) and conclude with a brief overview of WASP-C (§3.5).

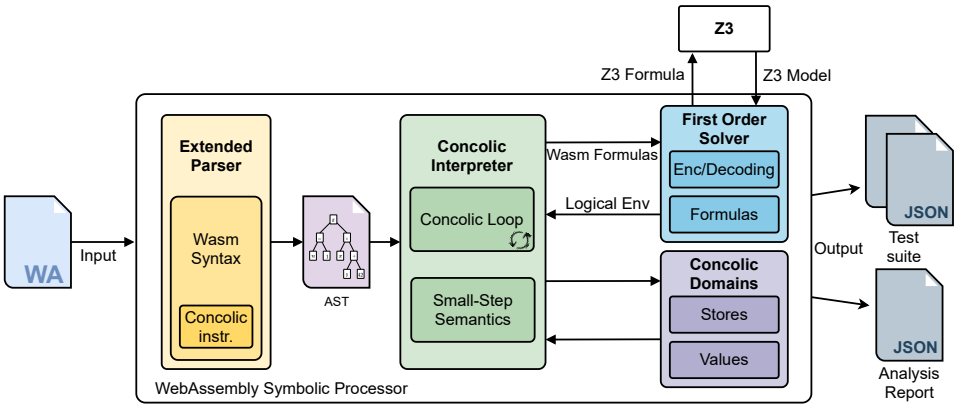


Figure 3 High-level architecture of WASP.

3.1 Overview

The goal of WASP is to explore multiple execution paths of the program to be analysed in order to uncover potential execution errors. To this end, the Wasm programs given to WASP must be annotated with first order assertions to be validated by WASP. WASP explores all the execution paths of the given program up to a pre-established depth. If no assertion failure is found, WASP provides a bounded verification guarantee. Otherwise, it outputs a concrete counter-model that triggers that failure.

WASP was developed on top of the Wasm reference interpreter [56], which we extended with symbolic facilities according to the high-level architecture described in Figure 3. In particular, we extended the original code-base of the reference interpreter with: **(1)** parsing facilities for the symbolic instructions required for declaring and reasoning over symbolic inputs; **(2)** a new concolic interpreter module, implementing the main concolic loop and the concolic execution of Wasm instructions; **(3)** a new concolic state module, implementing the main data structures we use to represent Wasm's concolic values, stacks, and memories; and **(4)** a dedicated first order solver used to encode the logic of WASP into the logic of its underlying SMT solver, Z3 [24].

Let us now take a look at how WASP concolically executes the Wasm program given as input. Firstly, the given program is parsed by our *Extended Parser*, generating an abstract syntax tree that is then passed to the *Concolic Interpreter*. The Concolic Interpreter implements the main concolic execution loop exploring one execution path at a time and generating for each path its corresponding path condition. The interpreter executes the given program by concolically evaluating one instruction at a time following the small-step concolic semantics presented in §3.2. Concolic execution requires the interpreter to keep track of both the program's concrete and symbolic states. To this end, we combine the concrete domains of the original reference interpreter with new symbolic domains modelling Wasm's symbolic values, stacks and memories. At the end of each concolic iteration, the Concolic Interpreter must interact with Z3 to determine the concrete values of the symbolic inputs for the next concolic iteration. This requires converting the logical formulas constructed by WASP into the logic of Z3. This is done by a dedicated *First Order Solver* that essentially translates WASP formulas into Z3 formulas using the Z3 OCaml bindings.

3.2 Concolic Execution Semantics

We define a concolic semantics of Wasm, which we use to guide the implementation of the concolic interpreter at the core of WASP. Our semantics operates on concolic states, which can be viewed as pairs of concrete and symbolic states. Concolic states are therefore inhabited by both concrete and symbolic values. Formally, symbolic values are given by the grammar:

$$\hat{s} := c \mid \hat{x} \mid \ominus(\hat{s}) \mid \oplus(\hat{s}, \hat{s}) \mid \otimes(\hat{s}, \hat{s}, \hat{s})$$

Symbolic values include: Wasm concrete values c, symbolic variables $\hat{x}$, and various Wasm unary and binary operators, respectively ranged by $\ominus$ and $\oplus$. Additionally, there is a ternary operator $\otimes$ reserved for symbolic byte expressions. As discussed above, we extended the syntax of Wasm with various instructions for creating and reasoning over symbolic values. In the formalism, we model the following three instructions:

$$e ::= \cdots \mid \texttt{sym\_assume} \mid \texttt{sym\_assert} \mid t.\texttt{symbolic}$$

Where: t.`symbolic` is used to create a symbolic value of type t; `sym_assume` is used to add the constraint on top of the stack to the current path condition; and `sym_assert` is used to check whether the constraint on top of the stack is implied by the current path condition.

Before proceeding to the description of the concolic semantics, we must first define concolic states. A concolic state is composed of: **(1)** a *concolic memory* μ, mapping integer addresses to pairs of concrete bytes and symbolic bytes; **(2)** a *concolic local store* ρ, mapping local variable indexes to pairs of concrete and symbolic values (e.g., $\rho = [0 \mapsto (2, \hat{y})]$); **(3)** a *concolic global store* δ mapping global variable indexes to pairs of concrete and symbolic values (e.g., $\delta = [0 \mapsto (2, \hat{y})]$); **(4)** a *concolic stack* st, consisting of a sequence of pairs of concrete and symbolic values (e.g., $st = (2, \hat{y}) :: (0, \hat{x})$); **(5)** a *symbolic environment* ε mapping symbolic variables to concrete values (e.g., $\varepsilon = [\hat{x} \mapsto 0, \hat{y} \mapsto 2]$); and **(6)** a *path condition* π keeping track of all the constraints on which the current execution has branched so far. All concolic domains are obtained by lifting the respective concrete domains, as defined in [30], from concrete values to pairs of concrete and symbolic values. For instance, while a concrete local store maps local variable indexes to concrete values, a concolic local store maps local variable indexes to pairs of concrete and symbolic values. In contrast to the concolic domains, symbolic environments do not have a counterpart in concrete execution. The concolic interpreter uses the symbolic environment to link the program's symbolic variables to their concrete values, essentially storing the bindings of the symbolic variables computed at the beginning of each concolic iteration.

The concolic semantics makes use of computation *outcomes* [26] to capture the flow of execution. We consider five types of outcomes: **(1)** the non-empty continuation outcome `Cont`(e), signifying that the execution of the current instruction generated a new instruction to be executed next; **(2)** the empty continuation outcome `Cont`, signifying that the execution may proceed to the next instruction; **(3)** the trap outcome `Trap`, signifying that the execution of the current instruction generated a Wasm trap; **(4)** the failed assertion outcome `AsrtFail`, signifying that the execution of the current instruction resulted in an assertion failure; and **(5)** the failed assumption outcome `AsmFail`, signifying that the execution of the current instruction resulted in an assumption failure. The concolic domains are summarised below.

Concolic Semantic Domains

Local Store	$\rho : i32 \to c \times \hat{s}$		
Stack	$st : (c \times \hat{s})\ \ list$	Outcome	$o ::=$ `Cont`(e) | `Cont` | `Trap` | `AsrtFail` | `AsmFail`
Logical Env	$\varepsilon : \hat{x} \to c$		
Path Cond	π	Symbolic Expr	$\hat{s} ::= c \mid \hat{x} \mid \ominus(\hat{s}) \mid \oplus(\hat{s}, \hat{s}) \mid \otimes(\hat{s}, \hat{s}, \hat{s})$
Global Store	$\delta : i32 \to c \times \hat{s}$		
Memory	$\mu : i32 \to c \times \hat{s}$		

$$\textsc{Load}\ \frac{n = \texttt{size}(t) \qquad (c, \hat{s}') = \texttt{load\_bytes}(\mu, k + o, n)}{t.\texttt{load}\ o, (k, \hat{s}) :: st, \mu\ \Rightarrow_{\mathsf{cs}}\ (c, \hat{s}') :: st, \mu, \texttt{Cont}}$$

$$\textsc{Store}\ \frac{\mu' = \texttt{store\_bytes}(\mu, k + o, (c, \hat{s}))}{t.\texttt{store}\ o, (k, \hat{s}_k) :: (c, \hat{s}) :: st, \mu\ \Rightarrow_{\mathsf{cs}}\ st, \mu', \texttt{Cont}}$$

$$\textsc{GetLocal}\ \frac{}{\texttt{get\_local}\ i, \rho, st\ \Rightarrow_{\mathsf{cs}}\ \rho, \rho(i) :: st, \texttt{Cont}}$$

$$\textsc{SetLocal}\ \frac{\rho' = \rho[i \mapsto (c, \hat{s})]}{\texttt{set\_local}\ i, \rho, (c, \hat{s}) :: st\ \Rightarrow_{\mathsf{cs}}\ \rho', st, \texttt{Cont}}$$

$$\textsc{SymAssert-CFail}\ \frac{c = 0}{\texttt{sym\_assert}, \rho, (c, \hat{s}) :: st, \varepsilon, \pi\ \Rightarrow_{\mathsf{cs}}\ \texttt{AsrtFail}}$$

$$\textsc{SymAssert-SFail}\ \frac{c \neq 0 \qquad (\pi \wedge (\hat{s} = 0))\ \mathsf{SAT}}{\texttt{sym\_assert}, (c, \hat{s}) :: st, \pi\ \Rightarrow_{\mathsf{cs}}\ st, \pi, \texttt{AsrtFail}}$$

$$\textsc{SymAssert-Pass}\ \frac{c \neq 0 \qquad (\pi \wedge (\hat{s} = 0))\ \mathsf{UNSAT}}{\texttt{sym\_assert}, (c, \hat{s}) :: st, \pi\ \Rightarrow_{\mathsf{cs}}\ st, \pi, \texttt{Cont}}$$

$$\textsc{SymAssume-Fail}\ \frac{st = (c, \hat{s}) :: st' \qquad c = 0 \qquad \pi' = \pi \wedge (\hat{s} = 0)}{\texttt{sym\_assume}, st, \pi\ \Rightarrow_{\mathsf{cs}}\ st', \pi', \texttt{AsmFail}}$$

$$\textsc{SymAssume-Pass}\ \frac{st = (c, \hat{s}) :: st' \qquad c \neq 0 \qquad \pi' = \pi \wedge (\hat{s} \neq 0)}{\texttt{sym\_assume}, st, \pi\ \Rightarrow_{\mathsf{cs}}\ st', \pi', \texttt{Cont}}$$

$$\textsc{Symbolic-Fresh}\ \frac{\hat{x} \notin \mathsf{dom}(\varepsilon) \qquad i \in t \qquad \varepsilon' = \varepsilon[\hat{x} \mapsto i]}{t.\texttt{symbolic}\ \hat{x}, st, \varepsilon\ \Rightarrow_{\mathsf{cs}}\ (i, \hat{x}) :: st, \varepsilon', \texttt{Cont}}$$

$$\textsc{Symbolic}\ \frac{\hat{x} \in \mathsf{dom}(\varepsilon)}{t.\texttt{symbolic}\ \hat{x}, st, \varepsilon\ \Rightarrow_{\mathsf{cs}}\ (\varepsilon(\hat{x}), \hat{x}) :: st, \varepsilon, \texttt{Cont}}$$

Figure 4 Fragment of WebAssembly concolic semantics: non-control-flow instructions.

We formalise the concolic semantics of Wasm instructions using a semantic judgement of the form: $e, \rho, st, \varepsilon, \pi, \delta, \mu\ \Rightarrow_{\mathsf{cs}}\ \rho', st', \varepsilon', \pi', \delta', \mu', o$ meaning that the concolic evaluation of the instruction e in the local store ρ, stack st, symbolic environment ε, path condition π, global store δ, and memory μ results in a new local store ρ', stack st', logical environment ε', path condition π', global store δ', memory μ', and outcome o. Figures 4 and 5 present a selection of the semantic rules. In the presentation of the rules, we omit the elements of the configuration that are neither updated nor inspected by the current rule, writing, for instance, $e, \rho, st\ \Rightarrow_{\mathsf{cs}}\ \rho', st', o$ to mean $e, \rho, st, \varepsilon, \pi, \delta, \mu\ \Rightarrow_{\mathsf{cs}}\ \rho', st', \varepsilon, \pi, \delta, \mu, o$. The selected concolic rules are explained below.

Load. This rule first computes the concrete address whose value is to be loaded from memory by adding the given offset parameter o to the concrete memory address k at the top of the stack. Then, the concolic pair stored at the real memory address $k + o$ is loaded from memory using the auxiliary function `load_bytes` and placed at the top of the stack. The function `load_bytes`, explained in §3.3, receives as parameter not only the memory μ and the concrete address $k + o$ but also the size n of the memory chunk to be loaded, which is determined using the auxiliary function `size`.

Store. This rule pops the first two concolic pairs out of the stack, with the second one denoting the value to be stored and the first one the memory address where to store it. Then, the rule computes the real memory address $k + o$ by adding the given offset parameter o to the concrete address k. Next, it uses the function `store_bytes`, explained in §3.3, for storing the concolic pair $(c, \hat{s})$ at $k + o$. In contrast to `load_bytes`, the function `store_btyes` does not require the size of the value to be stored which can be determined from the value c.

SymAssert. The SymAssert rules look at the value c on top of the stack $(c, \hat{s}) :: st$. If $c = 0$ (CFail), it immediately raises `AsrtFail` and the interpreter stops. Otherwise, it checks if that the current path condition implies that the the value on top of the stack is different

$$\textsc{If-True}\quad \frac{st = (c, \hat{s}) :: st' \qquad c \neq 0 \qquad \pi' = \pi \wedge (\hat{s} \neq 0) \qquad o = \texttt{Cont}\ (\texttt{block}\ tf\ e_1^*)}{\texttt{if}\ tf\ e_1^*\ \texttt{else}\ e_2^*, st, \pi\ \Rightarrow_{\mathsf{cs}}\ st', \pi', o}$$

$$\textsc{If-False}\quad \frac{st = (c, \hat{s}) :: st' \qquad c = 0 \qquad \pi' = \pi \wedge (\hat{s} = 0) \qquad o = \texttt{Cont}\ (\texttt{block}\ tf\ e_2^*)}{\texttt{if}\ tf\ e_1^*\ \texttt{else}\ e_2^*, st, \pi\ \Rightarrow_{\mathsf{cs}}\ st', \pi', o}$$

$$\textsc{Tbl-Brk-In}\quad \frac{st = (k, \hat{s}) :: st' \qquad \pi' = \pi \wedge (\hat{s} = k)}{\texttt{br\_table}\ j_1^k\ j\ j_2^*, st, \pi\ \Rightarrow_{\mathsf{cs}}\ st', \pi', \texttt{Cont}\ (\texttt{br}\ j)}$$

$$\textsc{Tbl-Brk-Out}\quad \frac{st = (c, \hat{s}) :: st' \qquad c \geq k \qquad \pi' = \pi \wedge (\hat{s} \geq k)}{\texttt{br\_table}\ j_1^k\ j, st, \pi\ \Rightarrow_{\mathsf{cs}}\ st', \pi', \texttt{Cont}\ (\texttt{br}\ j)}$$

$$\textsc{Call Indirect - Found}\quad \frac{st = (j, \hat{s}) :: st' \qquad \pi' = \pi \wedge (\hat{s} = j) \qquad funcs(j) = \left(\texttt{func}\ tf\ \texttt{local}\ t^*\ e^*\right)}{\texttt{call\_indirect}\ tf, st, \pi\ \Rightarrow_{\mathsf{cs}}\ st', \pi', \texttt{Cont}\ (\texttt{call}\ j)}$$

$$\textsc{Call Indirect - Not-Found}\quad \frac{st = (j, \hat{s}) :: st' \qquad j \notin \mathsf{dom}_{tf}(funcs) \qquad \pi' = \pi \wedge (\hat{s} \notin \mathsf{dom}_{tf}(funcs))}{\texttt{call\_indirect}\ tf, st, \pi\ \Rightarrow_{\mathsf{cs}}\ st', \pi', \texttt{Trap}}$$

Figure 5 Fragment of WebAssembly concolic semantics: control-flow instructions.

from 0; formally: $\pi \Rightarrow (\hat{s} \neq 0)$. Checking the validity of $\pi \Rightarrow (\hat{s} \neq 0)$ is equivalent to checking the satisfiability of $\neg(\pi \Rightarrow (\hat{s} \neq 0))$; formally: $\pi \Rightarrow (\hat{s} \neq 0)$ is valid *if and only if*, $\neg(\pi \Rightarrow (\hat{s} \neq 0))$ is **not** satisfiable. Simplifying $\neg(\pi \Rightarrow (\hat{s} \neq 0))$, we obtain the formula $\pi \wedge (\hat{s} = 0)$. Hence, the rule checks if the formula $\pi \wedge (\hat{s} = 0)$ is satisfiable, in which case (SFail) the assertion fails and the outcome `AsrtFail` produced; otherwise (Pass), the assertion holds and the program may continue, as given by the outcome `Cont`.

SymAssume. The SymAssume rules check the value c on top of the stack, which is expected to be different from 0. Hence, if $c = 0$ (Fail), the current concolic iteration can be discarded as it is not relevant to the programmer. To achieve this, the semantics leaves the current concolic state unchanged, generating the outcome `AsmFail` and extending the current path condition with the formula $\hat{s} = 0$. If $c \neq 0$ (Pass), the concolic execution may proceed, simply conjuncting the formula $\hat{s} \neq 0$ with the current path condition.

Symbolic. The Symbolic-Fresh and Symbolic rules are used for the creation of a symbolic variable of the type t, named $\hat{x}$. If the variable $\hat{x}$ is already present in the mappings of the symbolic environment, $\hat{x} \in \mathsf{dom}(\varepsilon)$, then this variable already exists and its mapped value is inserted on top of the stack $(\varepsilon(\hat{x}), \hat{x}) :: st$. If $\hat{x}$ does not exist in the symbolic environment a new entry is created, where $\hat{x}$ is mapped to a random value i of type t, resulting in the new symbolic environment $\varepsilon' = \varepsilon[\hat{x} \to i]$, and $(i, \hat{x})$ being put on top of the stack.

If. The If rule analyses the concrete value c on top of the stack $(c, \hat{s}) :: st$. If $c \neq 0$, then the path condition is conjoined with the symbolic expression associated with the value on top of the stack $\pi \wedge (\hat{s} \neq 0)$. The resulting outcome is a `block` with the set of instructions e_1^*, corresponding to the "then" branch. If $c = 0$, the opposite happens, the resulting path condition is $\pi \wedge (\hat{s} = 0)$, and the outcome is a block with the set of instructions e_2^*, corresponding to the "else" branch.

Table-Break. The Tbl-Brk rules first check the integer value k at the top of the stack and then inspect the list, $j_1, \ldots, j_n$, of argument indices. If $k \leq n$ (Table-Break-In), the semantics simply obtains the $(k{+}1)$-th index, j_{k+1}, and returns the outcome `Cont` (`br` j_{k+1}) to transfer the control to the j_{k+1} enclosing block. If $k > n$ (Table-Break-Out), the semantics proceeds as in the previous case but with the index j_n. For instance, the execution of (`br_table` $4, 3, 2, 1$) with the integer 2 on top of the stack generates the outcome `Cont` (`br` 2), while its execution with 7 on top of the stack generates `Cont` (`br` 1). At the symbolic level, both rules extend the path condition with information about index taken. The concolic semantics of `br` follows directly from its concrete semantics given in [30] as this rule does not interact with the symbolic elements of the concolic state.

Algorithm 1 Concolic interpreter main loop.

```
1 function ConcolicExecute(e, ρ, st, δ, μ)
2     Π, π ← true, true
3     while Π is SAT ∧ belowLimit() do
4         ε_i ← Model(π)
5         e, ρ, st, ε_i, π, δ, μ ⇒*_cs ρ', st', ε', π', δ', μ', o
6         if o = Error then
7             return false
8         Π ← Π ∧ ¬π'
9     return true
```

Call-Indirect. The Call Indirect rules first check the integer index j at the top of the stack and inspect the function table to obtain the corresponding function. The Found rule models the case in which the j-th function exists and its type coincides with the one provided to the call instruction. In this case, the semantics simply generates the outcome `Cont` (`call` j), indicating that the j-th function is to be executed next with no extra check. The Not-Found rule models the case in which either the j-th function does not exist or its type does not coincide with the given argument. In this case, the semantics generates the `Trap` outcome. At the symbolic level, both rules extend the current path condition. The Found rule is straightforward, simply recording the index of the executed function. The Not-Found rule is slightly more convoluted. Instead of simply recording the failing index ($\hat{s} = j$), it records all possible failing indexes ($\hat{s} \notin \mathsf{dom}_{tf}(funcs)$), preventing the concolic loop from generating concrete inputs that lead to new illegal calls at that execution point. Analogously to the `br` instruction, the concolic semantics of `call` also follows directly from its concrete semantics given in [30].

3.2.1 Concolic Loop

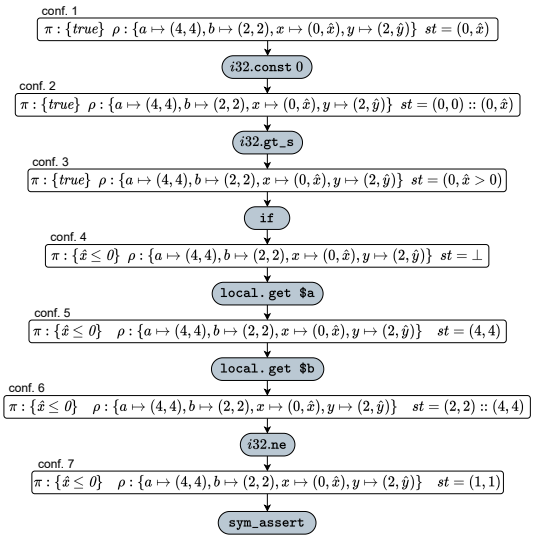
Concolic execution engines execute a given program multiple times in order to explore all possible execution paths. Algorithm 1 presents WASP's main concolic loop. To generate new concrete inputs at the end of each concolic iteration, the concolic interpreter maintains a *global path condition* Π representing all the execution paths that remain to be explored. At the beginning of each concolic iteration, the satisfiability of the global path condition is checked with the help of Z3. If Π is satisfiable, Z3 returns a model, which the engine uses to construct a new symbolic environment, mapping the symbolic variables of the program being analysed to new concrete values. If Π is not satisfiable, the execution stops, given that all possible execution paths have already been explored. Initially, Π is set to *true*, meaning that all paths still have to be explored. At the end of each iteration, the engine updates the global path condition to $\Pi \wedge \neg\pi'$, where π' is the final path condition of the iteration at hand. By adding $\neg\pi'$ to the global path condition, we prevent that future concolic iterations go down the same execution path as the current iteration.

3.2.2 Concolic Execution Example

To illustrate how the proposed concolic semantics works in practice, let us consider the Wasm program given in Figure 6a. This program results from the compilation of the C program given in Figure 2a. For clarity, we represent the given program in Wasm Textual Format (WAT), in which program variables are associated with string identifiers instead of

```
1  (func $main
2    ;; init x=symb(), y=symb(),
3    ;;      a=4, b=2...
4    (i32.const 0)
5    (i32.gt_s)
6    (if
7      (then
8        (i32.const 6)
9        (local.set $b)
10       (local.get $x)
11       (local.get $y)
12       (i32.lt_s)
13         (if
14           (then
15             (local.get $x)
16             (i32.const 2)
17             (i32.mul)
18             (local.get $y)
19             (i32.add)
20             (local.set $a)))))
21   (local.get $a)
22   (local.get $b)
23   (i32.ne)
24   (sym_assert))
```

(a) Wasm program for Listing 2a.



(b) Concolic execution flow diagram for the program in Listing 6a, executing the sequence of instructions indicated in lines: 4-6, and 21-24.

Figure 6 Concolic execution example in WASP.

integer indices. Figure 6b illustrates the concolic iteration corresponding to the right-most path of the execution tree in Figure 2b, with each node in the flow diagram representing a configuration of the concolic semantics. We represent the path condition π, local store ρ, execution stack st, and the current instruction to be executed. The symbolic environment, linear memory, and global store are not represented as they are not manipulated by the program. Specifically, the execution depicted in Figure 6b represents a concolic iteration of the program where the symbolic variables x and y are concretely assigned to 0 and 2, respectively. In this case, the first if-statement of the corresponding C program (i.e., $x > 0$) is false, causing the program to jump to the final assert instruction which holds (i.e., $a \neq b$). In the compiled Wasm program listed in Figure 2a, this if-statement corresponds to the Wasm instructions shown in lines 4-6. The concolic execution of these instructions leads to the first three configurations depicted in the flow diagram (confs. 1-3). The final assertion is translated to the lines 21-24 of the Wasm program, whose execution corresponds to the last four configurations in the flow diagram (confs. 4-7). At the end of the concolic iteration, WASP negates the obtained path condition, $\pi = \hat{x} \leq 0$, in order to generate the inputs for the next concolic iteration as explained in the previous sub-section.

3.3 Symbolic Memory

To concolically execute Wasm code, WASP requires the ability to reason at the byte-level granularity. This requirement is important because Wasm code often needs to operate over the in-memory representation of data at the finer-grained level of bytes or bits. One such example is given in Listing 7b, which shows a real-world function for converting the endianness of a 32-bit unsigned integer. To help explain the example, let us first consider the corresponding C function given in Listing 7a. This example receives an unsigned integer parameter `x` and returns the unsigned integer obtained by swapping the order of the bytes of `x`. Before proceeding to the description of the example, recall that: **(1)** the union data

```c
unsigned int swap(unsigned int x) {
  union {
    unsigned int i; char c[4];
  } src, dst;
  src.i = x;
  dst.c[3] = src.c[0];
  dst.c[2] = src.c[1];
  dst.c[1] = src.c[2];
  dst.c[0] = src.c[3];
  return dst.i;
}
```

(a) Endianness swap function, taken from [46].

```
(func $swap (param $x i32) (result i32)
  (local $src i32) (local $dst i32)
  (local.get $src)
  (local.get $x)
  (i32.store)
  (local.get $dst)
  (local.get $src)
  (i32.load8_u offset=0)
  (i32.store8  offset=3)
  ;; ...
  (return))
```

(b) Snippet of the Wasm program resulting from the compilation of the program in Listing 7a.

Figure 7 Symbolic byte manipulation example.

type is used for storing different data types in the same memory segment; **(2)** in a standard 32-bit architecture, characters are represented by one byte and integers by four bytes; and **(3)** local variables are stored in the stack segment of the C memory.

The `swap` function first declares two variables `src` and `dst`, which can hold either an unsigned integer or an array of four characters. Note that, the two members of this union take exactly the same space, 4 bytes. Then, it copies the four bytes of `x` to the segment of memory referenced by `src`. Next, it copies each individual byte of `src` to the segment of memory referenced by `dst` in reverse order; that is the last byte of `src` will be the first byte of `dst` and so on and so forth. Finally, the function returns the integer value of `dst`.

Note that, the same segment of memory is accessed differently depending on the member of the union type that is used to interact with it. If one uses the union member `i`, one reads/writes four bytes from/into the corresponding memory segment. Conversely, if one uses the union member `c`, one reads/writes a single byte from/into the corresponding memory segment. This example clearly demonstrates the need for byte-level reasoning in SE tools.

Byte-Level Operators. In order to reason about byte-level memory operations, we make use of the operators `concat` and `extract`, which work as follows:

- The expression $\texttt{concat}(\hat{s}_1, \hat{s}_2)$ denotes the bit-vector resulting from the concatenation of the bit-vectors denoted by $\hat{s}_1$ and $\hat{s}_2$. For instance, $\texttt{concat}(\texttt{0xBE}, \texttt{0xEF}) = \texttt{0xBEEF}$.
- The expression $\texttt{extract}(\hat{s}, h, l)$ denotes the bit-vector corresponding to the bytes of the bit-vector denoted by $\hat{s}$ that occur in $[l, h[$. This means that the expression $\texttt{extract}(\hat{s}, h, l)$ denotes a bit-vector of size $h - l$. For instance, $\texttt{extract}(\texttt{0xBEEF}, \texttt{1}, \texttt{0}) = \texttt{0xEF}$.

Given that our underlying Z3 encoding represents all primitive types as bit-vectors, the encoding of these operators into the logic of Z3 is trivial as they have equivalent Z3 operators.

Byte-Addressable Memory. Note that our concolic Wasm memory is a mapping from integer indexes, representing concrete memory addresses, to pairs of concrete and symbolic bytes. This means that before we store a given value in memory, we have to obtain the expressions denoting its corresponding bytes. Conversely, when loading a primitive type from memory, we must concatenate the symbolic expressions denoting its component bytes to obtain the symbolic expression that denotes the full value. To do this, we enlist two helpers:

- The function $\texttt{store\_bytes}(\mu, l, (c, \hat{s}))$, that individually unpacks each concrete and symbolic byte from the value pair $(c, \hat{s})$, using the `extract` operator, and then sequentially

stores the obtained concrete and symbolic bytes into the segment of μ pointed to by the l, resulting in a new symbolic memory μ'.

- The function $\texttt{load\_bytes}(\mu, l, n)$, that sequentially loads n concrete and symbolic bytes from the concolic memory at address l and concatenates them using the `concat` operator, resulting in a new concolic pair of the form $(c, \hat{s})$.

We mathematically formalise the functions `store_bytes` and `load_bytes` in the table below.

Memory Operations

$$\textsc{StoreBytes}\quad \frac{n = |c| \qquad c_i = \texttt{extract}(c, i, i-1)|_{i=1}^{n} \qquad \hat{s}_i = \texttt{extract}(\hat{s}, i, i-1)|_{i=1}^{n}}{\texttt{store\_bytes}(\mu, l, (c, \hat{s})) = \mu\left[l + (i-1) \mapsto (c_i, \hat{s}_i)\right]|_{i=1}^{n}}$$

$$\textsc{LoadBytes}\quad \frac{(c_i, \hat{s}_i) = \mu(l + (i-1))|_{i=1}^{n} \qquad c = \texttt{concat}(c_1, \ldots, c_n) \qquad \hat{s} = \texttt{concat}(\hat{s}_1, \ldots, \hat{s}_n)}{\texttt{load\_bytes}(\mu, l, n) = (c, \hat{s})}$$

Byte-level Simplifications. While the concrete application of the operators `extract` and `concat` always yields a fully resolved concrete value, it is often not possible to resolve the application of these operators to symbolic values. For instance, the application of `concat` to two symbolic values $\hat{s}_1$ and $\hat{s}_2$ simply yields the symbolic expression $\texttt{concat}(\hat{s}_1, \hat{s}_2)$. As every time WASP interacts with the heap, it applies byte-level operators to the values being stored or loaded, concolic execution rapidly increases the complexity of the symbolic expressions handled by the program. This constitutes a serious problem as the additional complexity introduced by byte-level operators is detrimental to the overall performance of WASP. To counter this issue, we apply two simple algebraic simplifications to symbolic values, every time a symbolic value is loaded from memory. The simplification rules are captured by the following algebraic identities:

$$h - l = \texttt{size}(\texttt{type}(\hat{s})) \Rightarrow \texttt{extract}(\hat{s}, h, l) = \hat{s} \tag{1}$$

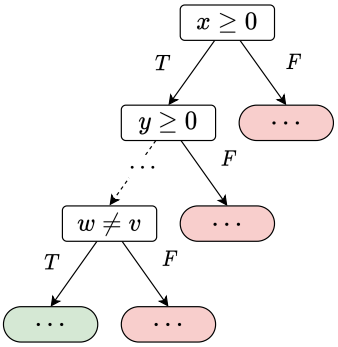
$$\texttt{concat}(\texttt{extract}(\hat{s}, h, m), \texttt{extract}(\hat{s}, m, l)) = \texttt{extract}(\hat{s}, h, l) \tag{2}$$

3.4 Shortcut Restarts

Programmers often need to test their functions not for all inputs but only for those that satisfy a specific set of constraints. In WASP, this can be done using the `sym_assume` instruction, which filters out all execution paths for which the symbolic inputs do not satisfy the given constraints. As explained in §3.2, whenever the symbolic interpreter encounters an assume statement whose constraint does not hold, it discards the current concolic iteration as it is not relevant for the developer. This design is, however, inherently inefficient as it requires WASP to restart the concolic execution of the program every time an assumption fails. To help understand this problem, let us consider the C program given in Figure 8a. This program starts with a sequence of n assumptions over its five symbolic variables. In the worst-case scenario, where every assumption fails, WASP would have to restart the analysis n times before actually starting executing the program. As a result, WASP would have to execute $O(n^2)$ lines of code and query Z3 n times before reaching the first meaningful concolic iteration, as illustrated by the execution tree given in Figure 8b.

```
int function_test() {
    int x = symb(), y = symb();
    int u = symb(), v = symb();
    int w = symb();
    // Setup assumptions (Asm)
    sym_assume(x >= 0);    // Asm_1
    sym_assume(y >= 0);    // Asm_2
    ...
    sym_assume(w != v);    // Asm_n
    // Program starts here
    ...
}
```

(a) Chained assumptions that do not affect the path condition set.



(b) Execution tree for the example in Listing 8a, where the red nodes imply a restart in WASP.

Figure 8 Example of assumption handling in WASP.

To solve this problem, we propose an adaptation of the concolic semantics of SymAssume given in Figure 4, which avoids the need for restarting the concolic execution from the beginning of the program whenever a failed assumption is reached. The concolic semantics of the assume instruction is captured by the rules given and described below.

Optimised SymAssume Semantic Rules

$$\textsc{SymAssume-Fail}\ \frac{c = 0 \qquad (\pi \wedge \hat{s})\ \mathsf{UNSAT}}{\texttt{sym\_assume}, ((c, \hat{s}) :: st), \pi \ \Rightarrow_{\mathsf{cs}}\ \rho, (\neg\hat{s} \wedge \pi), \texttt{AsmFail}} \qquad \textsc{SymAssume-Pass1}\ \frac{c \neq 0}{\texttt{sym\_assume}, ((c, \hat{s}) :: st), \pi \ \Rightarrow_{\mathsf{cs}}\ st, (\hat{s} \wedge \pi), \texttt{Cont}}$$

$$\textsc{SymAssume-Pass2}\ \frac{c = 0 \qquad \varepsilon' = \texttt{model}(\pi \wedge \hat{s}) \qquad (\rho', st', \delta', \mu') = \texttt{update\_model}(\varepsilon', (\rho, st, \delta, \mu))}{\texttt{sym\_assume}, \rho, ((c, \hat{s}) :: st), \varepsilon, \pi, \delta, \mu \ \Rightarrow_{\mathsf{cs}}\ \rho', st', \varepsilon', (\hat{s} \wedge \pi), \delta', \mu', \texttt{Cont}}$$

SymAssume-Fail. This rule is analogous to its previous version given in Figure 4. The difference is that now the current concolic execution is only terminated if there is no model for the conjunction of the current path condition and the formula being assumed, $\pi \wedge \hat{s}$. In this case, the current execution is incompatible with the assumed formula, meaning that it must be discarded.

SymAssume-Pass1. This rule is identical to its previous version given in Figure 4.

SymAssume-Pass2. This rule is the core of our proposed optimisation. It is applied when the current concrete execution does not satisfy the formula being assumed but the current path condition, π, is compatible with the assumed formula, $\hat{s}$. In this case, WASP queries Z3 for a model for $\pi \wedge \hat{s}$ and uses this model to build a new symbolic environment, ε', that satisfies the assumption. Then, WASP has to update all the concolic domains of the program in order for them to be consistent with the new symbolic environment, ε'. To this end, we make use of a function `update_model` that receives as input a symbolic environment ε' and a concolic state, generating a new concolic state, obtained by updating the concrete values of the input state according to the supplied symbolic environment.

3.5 WASP-C

WASP can also be adopted as a tool for indirectly analysing C programs. This section presents WASP-C, a symbolic execution framework to test C programs using WASP. WASP-C takes as input C programs annotated with assumptions and assertions and outputs a *test suite*. A

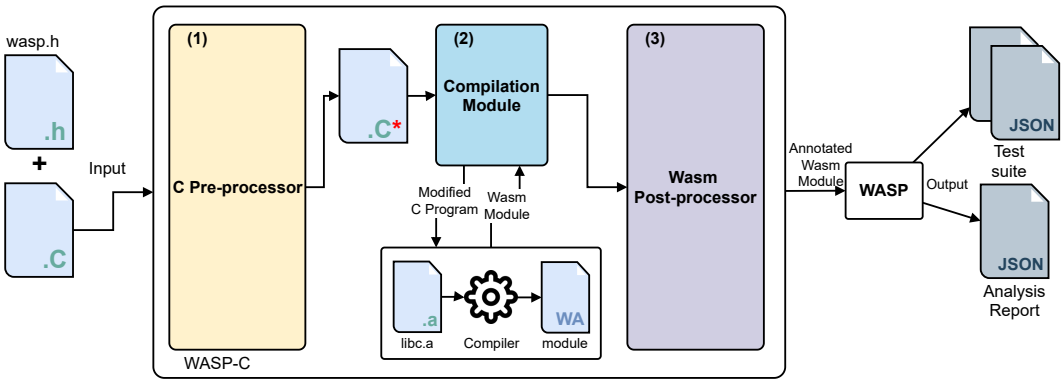


Figure 9 WASP-C high-level architecture.

test suite is a list of test cases, each corresponding to a JSON file, mapping the symbolic variables in the test to their corresponding concrete values. Each test case captures a different execution path of the program to be analysed. Since WASP does not directly operate over the C source code, WASP-C is comprised of three modules whose end goal is to generate a Wasm program for WASP to analyse.

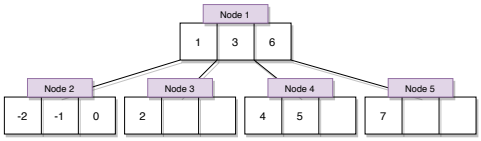
WASP-C is implemented in python and is composed of three essential modules: a *C Pre-processor*, a *Compilation Module*, and a *Wasm Post-processor*, which interact with each other according to the high-level architecture described in Figure 9. Using WASP as a submodule, WASP-C concolically executes C programs as follows. First, the *C Pre-processor* parses the given program using a standard C parser called *pycparser* [8], generating an abstract syntax tree (AST) that is then sent to a specialised C visitor (step 1). Our specialised C visitor traverses the AST, replacing binary operators such as logical ANDs and ORs with specific function calls to avoid spurious branching. Then the AST is exported back to a C program, which is subsequently compiled into Wasm by the *Compilation Module* (step 2). Lastly, the *Wasm Post-processor* processes the obtained Wasm module so as to inject the appropriate WASP symbolic primitives (step 3).

4 Evaluation

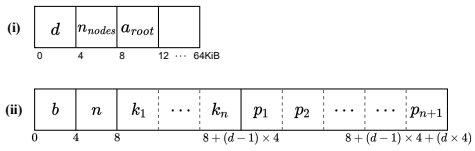
We evaluate WASP with respect to five evaluation questions: (EQ1) How does WASP compare to the existing symbolic execution tools for Wasm? (EQ2) Can WASP-C be used to detect bugs in C data structures? (EQ3) Can WASP-C support different types of symbolic reasoning? (EQ4) What are the performance gains of our proposed optimisation techniques? and (EQ5) Can WASP-C scale to industry-grade code?

When it comes to EQ1, we compare WASP against Manticore [51] as it is the only symbolic execution tool that can directly be applied to Wasm binaries; the other existing tool, WANA [73], works only on EOSIO and Ethereum smart contracts, not including a stand-alone symbolic execution engine for Wasm that can be run on its own.

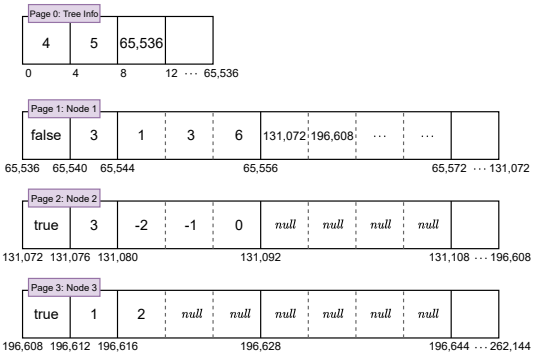
All experiments were performed on a server with a 12-core Intel Xeon E5–2620 CPU and 32GB of RAM running Ubuntu 20.04.2 LTS. For the constraint solver, we employed Z3 v4.8.1. For compiling our benchmarks, we used clang v10.0.0 as part of the LLVM compiler toolchain kit v10.0.0, which includes: *opt*, the LLVM optimiser and analyser; *llc*, the LLVM static compiler; and *wasm-ld*, the Wasm version of *lld*, which is the LLVM object linker. For each execution of WASP, we use the flag `-u` which disables WASP's type checker, set a timeout of 15 minutes, and limit the executing process to 15GiB of memory.



(a) Example of a B-tree with $t = 4$.



(b) Memory layout of a fully filled node ($n = d-1$). **(i)** B-tree metadata: degree, no. of nodes, and root address. **(ii)** B-tree node: leaf node b, no. of keys, key values, and children addresses.



(c) Memory layout of the first three nodes in Figure 10a.

Figure 10 Memory layout of a simple B-tree.

4.1 EQ1: Comparison with Manticore

To compare WASP with Manticore, we use both these tools to symbolically analyse a custom-made Wasm implementation of a B-tree data structure developed by C. Costa [22] and inspired by that of Watt et al. [75]. This data structure allows us to effectively test the scalability of both engines with respect to code size ($\approx$ 4000 LoC) and the complexity of the generated formulas. In the following, we first give a high-level description of the B-tree implementation and the experimental set-up and then present the obtained results.

B-Tree Implementation. B-trees are n-ary self-balancing trees typically used in the implementation of storage systems [55]. B-trees have a fixed branching factor d, denoting the maximum number of children that internal nodes may have. B-tree nodes store at most $d-1$ keys; internal nodes additionally store the pointers to their respective children. Intuitively, it is as if each internal node stores one key in between each two consecutive child pointers. The keys stored inside a B-tree are arranged so that: **(1)** the keys of every node are ordered; and **(2)** each key k_i stored in between the pointers p_i and p_{i+1} of an internal node is greater than all the keys stored in the node pointed to by p_i and less than those stored in the one pointed to by p_{i+1}. B-trees must additionally satisfy various other invariants; the reader is referred to [21] for a thorough account of B-trees and their properties. Figure 10a shows a B-tree with branching factor $d = 4$. The tree contains one internal node (Node 1) and four leaf nodes (Nodes 2 to 5), with the internal node storing three keys. Observe that, for instance, the second key stored in Node 1 (key 3) is greater than the single key stored in Node 3 (key 2) and less than both keys stored in Node 4 (keys 4 and 5).

The B-tree implementation we use [22], as that of Watt et al. [75], only holds 32-bit integer keys. Each B-tree node is kept in a separate memory page according to the memory layout given in Figure 10b. Each memory page stores: **(1)** a flag b, indicating if it represents a leaf node; **(2)** an integer n, denoting the number of keys that the node holds; and **(3)** n keys, $k_1, \ldots, k_n$. Additionally, each internal node stores $n+1$ child pointers, $p_1, \ldots, p_{n+1}$. The implementation uses an extra memory page for keeping metadata information about the B-tree, namely: its branching factor d, number of nodes n_{nodes}, and address of the root node a_{root}. Figure 10c shows the memory layout of the first three nodes of the B-tree presented in Figure 10a together with the extra memory page used to store its meta-information. The B-tree implementation comes with four main functions: **(1)** `$createBTree`(d) for creating an empty B-tree with the specified degree; **(2)** `$insertBTree`(t, k) for inserting the key k into the tree t; **(3)** `$searchBTree`(t, k) for checking if the tree t holds the key k; and **(4)** `$deleteBTree`(t, k) for deleting the key k from the tree t.

Table 1 Results WASP and Manticore applied to our B-tree benchmarks.

	$n_u = 1$				$n_u = 2$				$n_u = 3$			
n_o	n_{paths}	T_{WASP} (s)	T_{Mcore} (s)	$\times T_{WASP}$	n_{paths}	T_{WASP} (s)	T_{Mcore} (s)	$\times T_{WASP}$	n_{paths}	T_{WASP} (s)	T_{Mcore} (s)	$\times T_{WASP}$
2	3	0.14	2.69	×19.2	12	1.19	22.78	×19.1	60	15.54	260.95	×16.8
3	4	0.55	6.31	×11.5	20	5.04	77.13	×15.3	120	47.52	802.42	×16.9
4	5	2.10	11.29	×5.4	30	8.79	170.04	×19.3	210	137.14	1,886.55	×13.8
5	6	1.45	18.95	×13.1	42	16.41	340.32	×20.7	336	286.15	4,041.37	×14.1
6	7	2.40	35.65	×14.8	56	29.05	627.98	×21.6	504	696.35	8,046.52	×11.6
7	8	7.11	54.61	×7.7	72	51.09	1,161.62	×22.7	720	2,003.00	15,803.34	×7.9
8	9	6.90	90.63	×13.1	91	74.53	1,948.36	×26.1	–	–	–	–
9	10	11.18	133.68	×12.0	110	113.74	2,976.56	×26.2	–	–	–	–

Experimental Setup. In order to compare the performance of WASP against that of Manticore, we use the symbolic test suite of [22]. All symbolic tests follow the same code template but use a varying number of symbolic values, of which some are constrained to be ordered. In the following, we use n_o and n_u to denote respectively the number of ordered and unordered symbolic values used in each test.

Results. Table 1 presents the results obtained when running our symbolic test suite with WASP and Manticore. The number of ordered symbolic values used by the tests varies between 2 and 9 and the number of unordered values between 1 and 3; i.e., $2 \leq n_o \leq 9$ and $1 \leq n_u \leq 3$. For each pair (n_o, n_u), we provide: the number of explored paths (n_{paths}); the execution time for WASP (T_{WASP}); the execution time for Manticore (T_{Mcore}); and the WASP speed-up with respect to Manticore. As expected, the number of explored paths increases exponentially with the number of unordered symbolic values. This is reflected in the time taken by both engines to complete the analysis. For instance, WASP takes more than 30 minutes to run the test with $n_o = 7$ and $n_u = 3$, while taking less than one minute to run the test with $n_o = 7$ and $n_u = 2$.

Most significantly, from Table 1, we observe that WASP is consistently faster than Manticore, achieving a speed-up that ranges from 5.4× to 26.2× and averages 15.8×. We conjecture that WASP is able to outperform Manticore for two main reasons: **(1)** Manticore performs static symbolic execution, which means that it interacts more often with the underlying SMT solver and makes more intensive use of memory; and **(2)** Manticore is primarily written in Python which is significantly slower than OCaml.[1]

4.2 EQ2: Detecting Bugs in C Data Structures

To investigate whether WASP-C can detect bugs in complex C data structures, we used it to symbolically test *Collections-C*, a generic data structure library obtained from GitHub [4], which includes a variety of data structures, such as arrays, lists, ring buffers, and queues. In total, it implements ten different data structures spanning just over 11k LoC. The symbolic test suite we used to evaluate WASP on Collections-C comes from the Gillian project [26], in the context of which Collections-C was symbollically tested using Gillian-C, an instantiation of the Gillian framework for the C language. Gillian's authors developed a symbolic test suite that they run against Collections-C. This symbolic test suite consists of 161 symbolic test programs targeting the various data structure algorithms included in Collections-C.

Here, we test two different versions of Collections-C, a version with bugs previously found by the authors of Gillian-C,[2] henceforth *buggy version*, and the version resulting from the

[1] `http://roscidus.com/blog/blog/2014/06/06/python-to-ocaml-retrospective`

[2] Version with the 2 bugs identified by Gillian-C: `https://github.com/srdja/Collections-C/pull/119` and `https://github.com/srdja/Collections-C/pull/123`.

Table 2 Results for Gillian-C and WASP-C applied to corrected version of Collections-C.

Category	n_i	Baseline	WASP-C				
		T_{Gil} (s)	T_{WASP} (s)	T_{loop} (s)	T_{solver} (s)	*avg_paths*	$S\left(\frac{T_{Gil}}{T_{WASP}}\right)$
Slist	37	8.34	9.06	6.21	0.85	2	0.92
Pqueue	2	4.79	0.34	0.19	0.05	1	14.09
Stack	2	1.55	0.21	0.06	0.00	1	7.38
Deque	34	8.08	6.43	3.89	1.03	2	1.25
Array	21	7.00	7.00	5.41	1.44	5	1.00
Queue	4	2.11	1.99	1.69	0.18	4	1.06
RingBuffer	3	1.43	0.31	0.07	0.00	1	4.62
Treeset	6	7.07	4.89	4.43	1.43	7	1.45
Treetable	13	12.07	5.02	4.04	1.61	5	2.40
List	37	21.77	30.01	27.18	11.65	6	0.73
Total	159	74.21	65.26	53.17	18.24	34	1.14

correction of those two bugs,[3] henceforth *corrected version*. Essentially, we use WASP-C to execute 161 symbolic test programs developed in the context of the evaluation of the Gillian-C project both against the buggy and corrected version of Collections-C.

Experimental Procedure. We performed two experiments: in the first, we use Gillian-C and WASP-C to execute the symbolic test suite on the corrected version of Collections-C; and in the second, we used the two tools to execute the two error-triggering symbolic tests on the buggy version of Collections-C.

Experiment 1. Table 2 presents the results of Experiment 1, where we use both Gillian-C and WASP-C to test the corrected version of Collections-C. We present the obtained results for each data structure included in Collections-C, showing for each of them: the number of tests (n_i), the total execution time for Gillian-C (T_{Gil}), the total execution time for WASP (T_{WASP}[4]), the total time spent in the concolic interpreter (T_{loop}), the total time in the constraint solver (T_{solver}), the average number of paths explored (*avg_paths*), and the speedup between T_{Gil} and T_{WASP} (S). From the table we observe that, overall, WASP is 1.14× faster than Gillian-C at analysing the complete benchmark suite. And, in 7 out the 10 categories, WASP completes the program analysis faster than Gillian-C (i.e., $T_{WASP} < T_{Gil}$). We conjecture that this performance gain is due to WASP's analysis, i.e., Gillian-C performs static symbolic execution while WASP performs concolic execution, which is faster since it requires fewer interactions with the underlying solver.

During Experiment 1 WASP-C found a new heap-overflow bug in the Pqueue data structure. We confirmed the bug with a concrete test using *AddressSanitizer* [66], reported it to the developers, and fixed it via a pull request, which has already been accepted by the library's main developer.[5] The bug was caused by an integer overflow that subsequently leads to an array-out-of-bounds heap access. WASP-C is able to detect this bug because it models C integers using Z3 bit-vectors, whereas Gillian only used mathematical reals at the time of testing.[6]

Experiment 2. Table 3 presents the results of experiment two, where we use Gillian-C and WASP-C to test the two bug-triggering tests for the buggy version of Collections-C. Since

[3] Corrected version: `https://github.com/srdja/Collections-C`.

[4] Note that, $T_{WASP} = T_{loop} + T_{parse}$ and $T_{loop} = T_{solver} + T_{interpretation}$. Where the parsing and interpretation times, respectively T_{parse} and $T_{interpretation}$, were omitted from the table due to space

Table 3 Bug-finding statistics for Collections-C bugs by WASP and Gillian-C.

Test	Vulnerability	T_{Gil} (s)	T_{WASP} (s)	T_{loop} (s)	T_{solver} (s)	n_{paths}	S
`array_test_remove`	Found	1.40	0.20	0.08	0.03	1	7.00
`list_test_zipIterAdd`	Found	0.57	0.40	0.18	0.00	1	1.42
Total	2/2	1.97	0.60	0.26	0.03	2	3.28

there were only two tests, each triggering a different bug, each row in the table represents a different bug. For each bug, we indicate whether or not WASP found the bug; the remaining columns have the same meaning as in Table 2. As the table indicates, WASP-C is able to detect the bugs discovered by Gillian-C.

4.3 EQ3: Different Types of Symbolic Reasoning

To investigate our third evaluation question, we test WASP-C against the *Test-Comp benchmark suite* (Test-Comp) [10] and compare its results with those obtained for the testing tools submitted to the 2021 Test-Comp Competition [11]. The Test-Comp test suite is organised into different categories with each focusing on a different type of symbolic reasoning. For instance, the categories *Arrays*, *BitVectors* and *Loops* respectively aim at reasoning about array operations, bit-operations, and loops and recursion.

Test-Comp defines two types of testing tasks: **(1)** *Cover-Branches* tasks, whose goal is to generate a set of concrete tests that cover the greatest possible number of program branches, and **(2)** *Cover-Error* tasks, whose goal is to generate at least one set of inputs that lead the execution of the given program to an execution error.

Test-Comp defines a scoring system to classify testing tools depending on how well they perform on both types of tasks. Essentially, a tool is assigned three scores, one for each type of task and a global score. Below, we provide further details on the scoring system.

Experimental Setup. We separately evaluate WASP-C on the Cover-Branches and Cover-Error tasks. For each task, we assign WASP-C a global score computed as in Test-Comp. For Cover-Branches, the assigned score represents the coverage of the generated test suites. For Cover-Error, the assigned score represents the number of bugs found. For both tasks we present the results for each testing category (e.g., Arrays, BitVectors, ControlFlow, and so on). We obtain the global score for all analysed categories by applying a weighted average on the individual scores of each category according to the number of tests in that category.

Results. Table 4 presents the evaluation results per category for the Cover-Branches and Cover-Error tasks, respectively, and compares the results obtained for WASP-C with those obtained for the 11 tools submitted to Test-Comp 2021: FuSeBMC [2], CMA-ES Fuzz [40], CoVeriTest [12], HybridTiger [58], KLEE [14], Legion [48], LibKluzzer [44], PRTest [45], Symbiotic [18], TracerX [37], and VeriFuzz [7]. The table shows the minimum and maximum recorded scores obtained for the 11 submitted tools, and the score and rank obtained by WASP-C.[7] Furthermore, in order to better compare WASP-C's results with those of the

constraints.

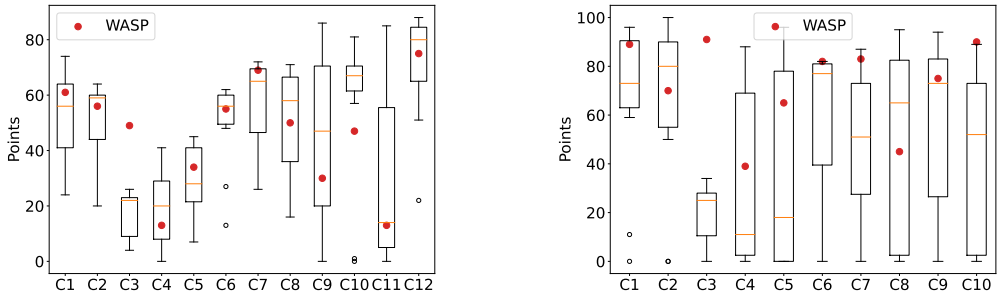
[5] Bug fix for heap-overflow bug: `https://github.com/srdja/Collections-C/pull/148`.

[6] Gillian has been since extended with support for mathematical integers and can now detect the bug.

[7] Cover-Error has no results for C11 and C12 because these categories have no Cover-Error tasks.

Table 4 Results for both meta categories: Coverage-Branches and Cover-Error.

	Cover-Branches				Cover-Error			
Category	Min	Max	WASP-C	Rank	Min	Max	WASP-C	Rank
C1.Arrays	96	296	245/380	4th	0	96	89/100	4th
C2.BitVectors	13	40	35/57	7th	0	10	7/10	2th
C3.ControlFlow	3	18	33/54	1st	0	11	29/32	1st
C4.ECA	0	12	4/27	8th	0	16	7/18	6th
C5.Floats	16	103	78/202	7th	0	32	21/32	5th
C6.Heap	19	90	80/136	7th	0	47	41/55	7th
C7.Loops	152	424	403/572	4th	0	138	127/156	4th
C8.Recursive	9	38	27/51	8th	0	19	9/20	7th
C9.Sequentialized	0	71	25/39	9th	0	101	75/107	9th
C10.XCSP	0	97	56/100	10th	0	53	54/59	1st
C11.Combinations	0	180	28/210	7th	–	–	–	–
C12.MainHeap	51	204	175/226	8th	–	–	–	–
Score	411	1389	1090	6th	0	405	360	3rd



(a) Box plot for Cover-Branches.

(b) Box plot for Cover-Error.

Figure 11 Box plots: Test-Comp coverage results.

other tools, Figures 11a and 11b plot the results, normalising the scores of all tools in each category and highlighting WASP-C's score with a red dot. For Cover-Branches and Cover-Error, WASP-C ranked sixth and third, respectively, ranking fourth overall. These results demonstrate that WASP-C's symbolic reasoning is on par with state-of-the-art symbolic execution and testing tools for C.

Finally, comparing the CPU time for the top 6 scoring tools in [11], WASP-C was the fifth-fastest tool in the Cover-Error category among these tools (with a total time of 26 hours) and the second-fastest tool in the Cover-Branches category (with a total time of 310 hours). Overall, WASP-C is the second-fastest tool among the top 6 scoring tools, finishing its analyses in about 326 hours and ranking fourth in terms of scoring (note that KLEE, the fastest tool, ranked sixth in terms of scoring). We further note that the tools we compare WASP-C against were executed on a superior testbed. Test-Comp's testbed is a computing cluster consisting of 168 machines; each test-generation run was executed on an otherwise wholly unloaded, dedicated machine to achieve precise measurements. Each machine had one Intel Xeon E3-1230 v5 CPU, with eight processing units each, a frequency of 3.4GHz and 33GB of RAM. This setting is superior to the one in which we run WASP-C: a single server with an Intel Xeon E5–2620 CPU, a frequency of 2.5GHz and 32GB of RAM.

Table 5 Experimental evaluation of OPTMem and OPTRestart.

Benchmark	T_{WASP} (h) on	T_{WASP} (h) off	T_{solver} (h) on	T_{solver} (h) off	n_{paths} on	n_{paths} off	n_{solver} on	n_{solver} off	avg_{cmds}(M) on	avg_{cmds}(M) off	no. ass. fls on	no. ass. fls off
OPTMem	32	71	1.80	0.04	16,196	261	15,975	26	603	7.6	106	3
OPTRestart	46	58	17.00	29.00	17,677	73,972	39,810	73,641	132	132	275	255

4.4 EQ4: WASP Optimisations

We introduce two optimisation techniques: **(1)** application of algebraic simplifications to byte-level symbolic expressions generated by memory interactions (§3.3); and **(2)** shortcut restarts for failed assumption statements (§3.4). In the following, we refer to the former technique as OPTMem and to the latter as OPTRestart. To investigate the effectiveness of these optimisations, we compare the execution times obtained for WASP-C when they are enabled (on) against those obtained when they are disabled (off).

Experimental Setup. For each optimisation technique, we have selected a subset of the 2021 Test-Comp benchmark suite [11] with the relevant features. For OPTMem, we have selected the four Test-Comp sub-categories with the highest number of calls to heap-manipulating functions (e.g. `malloc` and `calloc`), whereas for the OPTRestart, we have have selected the four sub-categories with the highest number of assume statements.

Results. Table 5 presents the results obtained for both optimisation techniques, showing: the total execution time of WASP (T_{WASP}); the total solver time (T_{solver}); the total number of explored paths (n_{paths}); the total number of calls to Z3 (n_{solver}); the average number of executed Wasm instructions (avg_{cmd}); and the total number of triggered assertion violations. The values of all metrics are given with the corresponding technique turned on and off.

With OPTMem on, WASP completes the analysis in less 39 hours (32 vs 71), explores 62 times more paths, and discovers 35 times more assertion violations. The main effect of this optimisation is to reduce the size of concolic states by reducing the size of the symbolic expressions that they store. This reduction enables WASP to interpret more instructions per unit of time ($5,234.6\ i/s$ vs $30.0\ i/s$), as it has been demonstrated that symbolic execution throughput is severely impacted by intensive memory usage [17].

With OPTRestart on, WASP completes the analyses in less 12 hours and discovers 20 more assertion violations. The main effect of this optimisation is that it allows WASP to explore fewer execution paths by ignoring the paths that lead to failed assumptions, thereby leaving more time for the exploration of relevant paths.

4.5 EQ5: Scalability to Industry-Grade Code

To investigate our fifth evaluation question, we use WASP-C to obtain a comprehensive test suite for part of the C implementation of the *AWS Amazon Encryption SDK* [67], a highly-used cryptographic library that powers, for instance, the Amazon DynamoDB Encryption Client [3]. The AWS Encryption SDK for C is a library for the encryption and decryption of data that implements complex data structures and algorithms in the C language. This library is challenging to analyse as it uses various cryptographic functions that current SMT solvers cannot tackle. The library comes with a benchmark suite of bounded verification proofs designed to be checked with the *CBMC bounded model checker* [42], which can be

Table 6 Benchmark results applying WASP-C to the AWS Encryption SDK for C.

Category	n_i	n_{paths}	T_{loop} (s)	T_{solver} (s)	T_{WASP} (s)	Coverage
Md	2	3	0.12	0.04	0.18	60.8
Decrypt	3	151	48.81	9.03	49.00	54.4
Edk	5	194	366.83	4.64	367.13	60.2
Cmm	5	558	1,793.00	60.43	1,794.00	66.6
Private	3	962	1,792.53	406.06	1,793.11	55.0
Keyring	10	1,382	1,145.89	213.67	1,146.56	70.8
Misc-ops	7	3,851	1,907.59	134.81	1,908.18	48.5
Total	35	7,101	7,054.77	828.68	7,058.16	59.5

easily turned into symbolic tests to enable the generation of a concrete test suite for the library. We consider 35 verification proofs totalling 2.3k LoC. The library itself contains multiple C files totalling just under 40k LoC.

Experimental Procedure. Our experimental procedures analyse the benchmark suite without constraining the number of paths explored during concolic execution and with a timeout of 15 minutes. We choose these settings to enable WASP to freely analyse every path of a program. Note that not all symbolic tests will take 15 minutes to be executed, as some tests do not loop on symbolic values and therefore have a finite execution tree.

Results. Table 6 presents the results of running the created symbolic test suite on seven modules of the AWS Encryption SDK for C organised by the data structure or algorithm that they are testing. Additionally, tests for generic data structures like lists or hash tables and generic operations like getters/setters go into the Misc-ops module as they are not specific to the encryption library. For each module, we present the number of tests targeting that module (n_i), the total number of paths explored (n_{paths}), the total time spent in the concolic loop (T_{loop}), the total time spent in Z3 (T_{solver}), the total analysis time (T_{WASP}), and the line coverage obtained. The table shows that, in total, WASP-C analyses the benchmark suite in just under two hours and obtains roughly 60% of line coverage of the library's functions. In contrast to the data structures in *Md*, which are mainly populated with concrete values and are therefore quickly analysed, the data structures *Edk*, *Cmm*, *Private*, and *Keyring* take a significant amount of time to be analysed as they mainly operate on symbolic values. Unsurprisingly, the symbolic tests in the former group trigger much fewer symbolic execution paths than those in the latter. Note that analyses finish quickly in the Decrypt module that tests decryption operations due to the small inputs given to these operations, typically, strings with one or two characters at most.

As our symbolic test suite is automatically obtained from the bounded verification proofs that come with the AWS Encryption SDK for C, its coverage is limited by the structure of the bounded inputs considered in the proofs. As most proofs only consider well-formed inputs, our symbolic test suite does not cover most of the library's code for handling ill-formed inputs. In the future, we plan to write additional tests to obtain 100% line coverage.

5 Related Work

Semantics of Wasm. Haas et al. [30] proposed a small-step operational semantics for Wasm together with a type system for checking the safety of stack operations. Later, Watt [74] mechanised both the semantics and the type system introduced in [30] using the Isabelle

theorem prover [53] and exposing several issues in the official Wasm specification. The authors of [75] then introduced Wasm Logic, a program logic for modular reasoning about heap-manipulating Wasm programs. In contrast to Wasm's native type system, Wasm Logic can be used to establish the safety of heap operations. However, it cannot yet reason about real-world Wasm code as it has not been automated. Very recently, Watt et al. [76] introduced two new mechanisations of the specification of Wasm following the new official W3C standard [72]; one developed in Isabelle and the other in the Coq [70] theorem prover.

Program Analyses for Wasm. Since the proposal of the Wasm standard [57], various program analyses have been designed for tackling the specificities of the language. Most of these analyses aim at the verification/testing of security properties and can broadly be divided into two main categories: *static analyses* [77, 68], which analyse stand-alone Wasm modules without the need to execute them, and *dynamic analyses* [69, 68], which instrument the given module to enforce the desired security property. Among the static analyses, we highlight:

- CT-Wasm [77], a type-driven extension of Wasm for provably secure implementation of cryptographic algorithms, which enforces information flow security and resistance to timing side-channel attacks through the use of security types;
- Wassail [68], an information flow analysis for Wasm based on a standard data-flow analysis, which the authors evaluate on a benchmark comprising 30 C programs.

Neither CT-Wasm nor Wassail can precisely reason about Wasm programs that interact with the memory, as they both assume that the values stored in memory are always confidential. In the future, we would like to study how to take advantage of WASP to improve the precision of information flow analysis for Wasm using, for instance, the self-composition technique [6] for the generation of vulnerability-triggering inputs.

Among the dynamic analyses for Wasm, we highlight the following two taint-tracking tools: TaintAssembly [27] and the tool presented in [69]. TaintAssembly [27] is a modification of the V8 JavaScript engine for performing basic taint tracking by adding a taint label to function parameters, local variables, and linear memory cells. In [69], the authors present a JavaScript virtual machine (VM) to interpret and run Wasm code, capable of monitoring the flow of sensitive information through taint tracking. However, neither TaintAssembly nor the JavaScript VM described in [69] can accurately track information flows in Wasm, as the former does not propagate indirect taint in comparison operators and the latter does not support floating-points in Wasm. Additionally, these tools are not ideal for testing generic Wasm code as they require concrete inputs to trigger the illegal information flows in the given program. A possible direction for future work is to combine WASP with a taint tracking tool, using WASP to generate inputs.

Symbolic Execution. Symbolic execution has been extensively used to find crucial errors and vulnerabilities in a broad spectrum of programming languages, such as C [28], C++ [14], Java [64], and Python [19]. Regarding the Web, there are several state-of-the-art tools for symbolically executing JavaScript code [60, 61, 62, 47, 65], demonstrating the need for such tools for the validation and testing of modern Web applications.

Symbolic execution tools can be divided into two main classes: *static* and *dynamic/concolic* [5]. Static symbolic execution engines, such as [41, 54, 39, 71, 60, 61], explore the entire symbolic execution tree up to a pre-established depth, while concolic execution engines, such as [14, 28, 64, 17, 62, 47, 65], usually work by pairing up a concrete execution with a symbolic execution and exploring one execution path at a time. An advantage of concolic execution over static symbolic execution is that concolic execution requires less frequent interactions

with the solver and a simpler memory model. There is a vast body of research on both static and concolic symbolic execution tools for a wide variety of programming languages, see [5, 15, 16] for comprehensive surveys on the topic. In the following, we give a detailed account of the only two existing symbolic execution tools for Wasm other than WASP.

WANA [73] is a cross-platform smart contract vulnerability detection tool employed to find vulnerabilities in EOSIO [43] and Ethereum smart contracts [38]. WANA is based on static symbolic execution and operates over Wasm bytecode. To detect vulnerabilities in smart contracts, WANA comes with three heuristics for EOSIO smart contracts and four for Ethereum smart contracts. Unlike WASP, WANA lacks a stand-alone symbolic execution engine for Wasm. Hence, it is not possible to run WANA on arbitrary Wasm code without refactoring its internal architecture. For this reason, we were unable to evaluate WANA on our B-tree implementation and compare its performance with those of WASP and Manticore.

Manticore [51] is a symbolic execution framework for binaries and smart contracts. Manticore is highly flexible, supporting a wide range of binaries and computing environments, including Wasm bytecode. When it comes to Wasm, Manticore does not expose dedicated primitives for constructing and reasoning over symbolic values at the source language level. Symbolic inputs and constraints are created as part of a complex Python script that must be written for each test [33], which initialises the symbolic state and calls the appropriate Wasm module. This process does not scale for a broad evaluation, as one would have to manually write a python script for each symbolic test. Nonetheless, we compare the performance of WASP with that of Manticore [51] in the analysis of a stand-alone Wasm implementation of a B-tree data structure, demonstrating that WASP is consistently faster.

6 Conclusion

In this paper, we presented WASP, a novel concolic execution engine for testing Wasm modules. To the best of our knowledge, WASP is the first symbolic execution tool to analyse complex WebAssembly code for general-purpose applications. Prior existing tools for symbolically executing Wasm code [51, 73] were only evaluated on smart contracts, which are simpler and therefore easier to analyse than general-purpose applications. On top of WASP, we also developed WASP-C, a symbolic execution framework to test C programs symbolically using WASP. We have extensively evaluated our tools. Our results show that WASP: **(1)** can detect bugs in complex data structure libraries, being able to find a previously unknown bug in a widely-used generic data structure library for C; **(2)** supports different types of symbolic reasoning, having comparable performance to well-established symbolic execution and testing tools for C; and **(3)** can scale to industry-grade code, being able to generate a high-coverage test suite for the Amazon Encryption SDK for C.

References

1 Syrus Akbary and Ivan Enderlin. Wasmer: Run any code on any client [online]. Accessed 27th-October-2021. URL: `https://wasmer.io`.

2 Kaled M. Alshmrany, Rafael S. Menezes, Mikhail R. Gadelha, and Lucas C. Cordeiro. FuSeBMC: A White-Box Fuzzer for Finding Security Vulnerabilities in C Programs (Competition Contribution). In *Fundamental Approaches to Software Engineering*, 2021.

3 AWS. Amazon DynamoDB Encryption Client [online]. Accessed 28th-October-2021. URL: `https://docs.aws.amazon.com/crypto/latest/userguide/awscryp-service-ddb-client.html`.

4 Sacha Ayoun, Alexis Marinoiu, and Petar Maksimović. Collections-C for symbolic testing with Gillian-C [online]. Accessed 15th-December-2020. URL: `https://github.com/GillianPlatform/collections-c-for-gillian` [cited 15th December 2020].

5 Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 2018.

6 Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 2011.

7 Animesh Basak Chowdhury, Raveendra Kumar Medicherla, and Venkatesh R. VeriFuzz: Program Aware Fuzzing. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2019.

8 Eli Bendersky. pycparser [online]. Accessed 1st-November-2021. URL: `https://github.com/eliben/pycparser`.

9 John Bergbom. Memory safety: old vulnerabilities become new with WebAssembly. Technical report, Forcepoint, December 2018.

10 Dirk Beyer. International Competition on Software Testing (Test-Comp). In *Tools and Algorithms for the Construction and Analysis of Systems*, 2019.

11 Dirk Beyer. Status Report on Software Testing: Test-Comp 2021. In *Fundamental Approaches to Software Engineering*, 2021.

12 Dirk Beyer and Marie-Christine Jakobs. CoVeriTest: Cooperative Verifier-Based Testing. In *Fundamental Approaches to Software Engineering*, 2019.

13 Ruben Bridgewate. Node v12.3.0 [online]. Accessed 27th-October-2021. URL: `https://nodejs.org/en/blog/release/v12.3.0`.

14 Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX Conference on Operating Systems Design and Implementation*, 2008.

15 Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S. Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *International Conference on Software Engineering*, 2011.

16 Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Communications of the ACM*, 2013.

17 Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing Mayhem on Binary Code. In *IEEE Symposium on Security and Privacy*, 2012.

18 Marek Chalupa, Jakub Novák, and Jan Strejcek. Symbiotic 8: Parallel and targeted test generation. *Fundamental Approaches to Software Engineering*, 2021.

19 Ting Chen, Xiao-song Zhang, Rui-dong Chen, Bo Yang, and Yang Bai. Conpy: Concolic execution engine for python applications. In *Algorithms and Architectures for Parallel Processing*, 2014.

20 Lin Clark. Standardizing WASI: A system interface to run WebAssembly outside the web [online]. Accessed 27th-October-2021. URL: `https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface`.

21 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

22 C. Costa. Concolic execution for WebAssembly. Master's thesis, Instituto Superior Técnico, Universidade de Lisboa, 2020. Master's Thesis.

23 Michael Pradel Daniel Lehmann, Johannes Kinder. Everything Old is New Again: Binary Security of WebAssembly. In *USENIX Security Symposium*, 2020.

24 Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

25 Jonathan Foote. Hijacking the control flow of a WebAssembly program [online]. Accessed 27th-October-2021. URL: `https://www.fastly.com/blog/hijacking-control-flow-webassembly` [cited 27th October 2021].

26 José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. Gillian, Part I: A Multi-Language Platform for Symbolic Execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

27 William Fu, Raymond Lin, and Daniel Inge. Taintassembly: Taint-based information flow control tracking for WebAssembly. *arXiv preprint*, 2018.

28 Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

29 GoogleSecurityResearch. Google Chrome 73.0.3683.103 - 'WasmMemoryObject::Grow' Use-After-Free [online]. Accessed 27th-October-2021. URL: `https://www.exploit-db.com/exploits/46968` [cited 27th October 2021].

30 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

31 Adam Hall and Umakishore Ramachandran. An execution model for serverless functions at the edge. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2019.

32 Paul Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems*, 1997.

33 Eric Hennenfent. Symbolically Executing WebAssembly in Manticore [online]. Accessed: 30th-November-2021. URL: `https://blog.trailofbits.com/2020/01/31/symbolically-executing-webassembly-in-manticore`.

34 Pat Hickey. Announcing Lucet: Fastly's native WebAssembly compiler and runtime [online]. Accessed 27th-October-2021. URL: `https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime`.

35 Aaron Hilbig, Daniel Lehmann, and Michael Pradel. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *Proceedings of the Web Conference*, 2021.

36 IEEE. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, 2019.

37 Joxan Jaffar, Rasool Maghareh, Sangharatna Godboley, and Xuan-Linh Ha. Tracerx: Dynamic symbolic execution with interpolation (competition contribution). *Fundamental Approaches to Software Engineering*, 2020.

38 Bhaskar Kashyap. Introduction to smart contracts [online]. Accessed: 30th-November-2021. URL: `https://ethereum.org/en/developers/docs/smart-contracts`.

39 Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

40 H. Kim. Fuzzing with stochastic optimization. Master's thesis, LMU Munich, 2020. Bachelor's Thesis.

41 James C. King. Symbolic execution and program testing. *Communications of the ACM*, 1976.

42 Daniel Kroening and Michael Tautschnig. CBMC – C Bounded Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2014.

43 Daniel Larimer and Brendan Blumer. EOS.IO Technical White Paper [online]. Accessed: 30th-November-2021. URL: `https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md`.

44 Hoang M. Le. LLVM-based Hybrid Fuzzing with LibKluzzer (Competition Contribution). In *FASE*, 2020.

45 Thomas Lemberger. Plain random test generation with PRTest. *International Journal on Software Tools for Technology Transfer*, 2020.

46 Xavier Leroy, Andrew W. Appel, Sandrine Blazy, and Gordon Stewart. The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA, June 2012. URL: `https://hal.inria.fr/hal-00703441`.

47 Guodong Li, Esben Andreasen, and Indradeep Ghosh. SymJS: Automatic symbolic testing of JavaScript web applications. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

48 Dongge Liu, Gidon Ernst, Toby Murray, and Benjamin I. P. Rubinstein. Legion: Best-First Concolic Testing (Competition Contribution). In *Fundamental Approaches to Software Engineering*, 2020.

49 Aishwarya Lonkar and Siddhesh Chandrayan. The dark side of WebAssembly [online]. Accessed 27th-October-2021. URL: `https://www.virusbulletin.com/virusbulletin/2018/10/dark-side-webassembly` [cited 27th October 2021].

50 Brian McFadden, Tyler Lukasiewicz, Jeff Dileo, and Justin Engler. Security Chasms of WASM. Technical report, NCC Group, August 2018.

51 Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts, 2019. `arXiv:1907.03890`.

52 Srđan Panić. Collections-C [online]. Accessed 5th-July-2021. URL: `https://github.com/srdja/Collections-C` [cited 5th July 2021].

53 Lawrence C Paulson. Isabelle [online]. Accessed 27th-November-2021. URL: `https://isabelle.in.tum.de`.

54 Corina S. Păsăreanu and Neha Rungta. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *IEEE/ACM International Conference on Automated Software Engineering*, 2010.

55 Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *ACM Transactions on Storage*, 2013.

56 Andreas Rossberg. WebAssembly Reference Interpreter [online]. Accessed 3rd-December-2020. URL: `https://github.com/WebAssembly/spec/tree/master/interpreter` [cited 3rd December 2020].

57 Andreas Rossberg. WebAssembly Core Specification. Technical report, W3C, 2019. URL: `https://www.w3.org/TR/wasm-core-1`.

58 Sebastian Ruland, Malte Lochau, and Marie-Christine Jakobs. HybridTiger: Hybrid Model Checking and Domination-based Partitioning for Efficient Multi-Goal Test-Suite Generation (Competition Contribution). In *Fundamental Approaches to Software Engineering*, 2020.

59 Jan Rüth, Torsten Zimmermann, Konrad Wolsing, and Oliver Hohlfeld. Digging into browser-based crypto mining. In *Internet Measurement Conference*, 2018.

60 José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa Gardner. Symbolic execution for JavaScript. In *International Symposium on Principles and Practice of Declarative Programming*, 2018.

61 José Fragoso Santos, Petar Maksimović, Gabriela Sampaio, and Philippa Gardner. JaVerT 2.0: compositional symbolic execution for JavaScript. *Proceedings of the ACM on Programming Languages*, 2019.

62 Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. In *IEEE Symposium on Security and Privacy*, 2010.

63 Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *CAV*, 2006.

64 Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. *ACM SIGSOFT Software Engineering Notes*, 2005.

65 Koushik Sen, George Necula, Liang Gong, and Wontae Choi. MultiSE: Multi-path symbolic execution using value summaries. In *Joint Meeting on Foundations of Software Engineering*, 2015.

66 Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Conference on Annual Technical Conference*, 2012.

67 Amazon Web Services. AWS Encryption SDK for C. `https://github.com/aws/aws-encryption-sdk-c`. Accessed: 2021-09-08.

68 Quentin Stiévenart and Coen De Roover. Compositional Information Flow Analysis for WebAssembly Programs. In *IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2020.

69 Aron Szanto, Timothy Tamm, and Artidoro Pagnoni. Taint tracking for WebAssembly. *arXiv preprint*, 2018.

70 The Coq Development Team. The Coq Proof Assistant [online]. Accessed 27th-November-2021. URL: `https://coq.inria.fr`.

71 Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

72 W3C. W3C WebAssembly Core Specification [online]. Accessed 27th-November-2021. URL: `https://www.w3.org/TR/wasm-core-1`.

73 Dong Wang, Bo Jiang, and W. K. Chan. WANA: Symbolic Execution of Wasm Bytecode for Cross-Platform Smart Contract Vulnerability Detection, 2020. `arXiv:2007.15510`.

74 Conrad Watt. Mechanising and verifying the WebAssembly specification. In *ACM SIGPLAN International Conference on Certified Programs and Proofs*, 2018.

75 Conrad Watt, Petar Maksimovic, Neelakantan R. Krishnaswami, and Philippa Gardner. A program logic for first-order encapsulated WebAssembly. In Alastair F. Donaldson, editor, *European Conference on Object-Oriented Programming*, 2019.

76 Conrad Watt, Xiaojia Rao, Jean Pichon-Pharabod, Martin Bodin, and Philippa Gardner. Two Mechanisations of WebAssembly 1.0. In *Formal Methods*, 2021.

77 Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. CT-Wasm: Type-driven Secure Cryptography for the Web Ecosystem. *Proceeding of the ACM on Programming Languages*, 2019.

78 WebAssembly. WebAssembly FAQ [online]. Accessed: 29th-October-2021. URL: `https://webassembly.org/docs/faq`.

79 Shu-Hung You, Robert Bruce Findler, and Christos Dimoulas. Sound and complete concolic testing for higher-order functions. In *ESOP*, 2021.