TÉCNICO LISBOA

Robust Symbolic Execution for WebAssembly

Filipe dos Santos Oliveira Marques

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. José Faustino Fragoso Femenin dos Santos
Prof. Nuno Miguel Carvalho Santos

Examination Committee

Chairperson: Prof. José Luís Brinquete Borbinha
Supervisor: Prof. José Faustino Fragoso Femenin dos Santos
Member of the Committee: Prof. Mikoláš Janota

October 2021

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

Acknowledgments

I would like to thank my parents for their friendship, encouragement and caring over all these years, for always being there for me through thick and thin and without whom this project would not be possible. I would also like to thank my grandparents, aunts, uncles and cousins for their understanding and support throughout all these years.

I would also like to acknowledge my dissertation supervisors Prof. José Fragoso and Prof. Nuno Santos for their insight, support and sharing of knowledge that has made this Thesis possible.

Last but not least, to all my friends and colleagues that helped me grow as a person and were always there for me during the good and bad times in my life. Thank you.

To each and every one of you – Thank you.

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 to introduce vulnerabilities into Web programs, as pre-existing vulnerabilities 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 security critical situation, there is still a 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. WASP was further tested against the Test-Comp benchmark, obtaining results comparable to well-established symbolic execution/testing tools for C, such as KLEE and VeriFuzz.

Keywords

Concolic Testing, WebAssembly, Test-Generation, Testing C Programs

Resumo

WebAssembly (Wasm) é um novo formato binário que permite que código escrito em linguagens de alto nível seja executado na Web, com velocidade quase nativa, pelo motor de JavaScript no browser. Apesar das suas óbvias vantagens, o Wasm abre oportunidades de introdução de vulnerabilidades em programas da Web, dado que as vulnerabilidades pré-existentes em programas escritos em linguagens inseguras podem ser transferidas para os binários de Wasm. Frequentemente, o código-fonte destes binários não se encontra disponível para análise estática, originando a necessidade de ferramentas que consigam trabalhar com código Wasm diretamente. Apesar desta necessidade crítica de segurança para binários Wasm, não existe uma ferramenta convencional para análise de Wasm. Nesta tese apresentamos o WASP, um motor de execução simbólica para testar módulos Wasm. O WASP funciona diretamente sobre código Wasm, sendo desenvolvido em cima de um interpretador de referência. O WASP foi avaliado cuidadosamente. Foi usado para testar simbolicamente uma biblioteca genérica de estruturas de dados em C e da Amazon Encryption SDK, também para C. Foi demonstrado que a ferramenta deteta erros e gera testes de alta cobertura para aplicações em C, no mundo real. O WASP foi ainda testado com a bateria de testes da Test-Comp, obtendo resultados comparáveis com ferramentas de execução simbólica para C, tais como o KLEE e o VeriFuzz.

Palavras Chave

Execução Concólica, WebAssembly, Geração de Testes, Verificação de Programas C

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

AWS	Amazon Web Services
SDK	Software Development Kit
WASP	WebAssembly Symbolic Processor
IO	Input/Output
LOC	lines of code
WAT	Wasm Textual Format
SMT	Satisfiability Modulo Theories

1

Introduction

WebAssembly (Wasm) is a binary instruction format designed to be the new standard compilation target for the Web. Wasm is now either partially or fully 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 for uses in server-side runtimes[1], IoT platforms [4], and in edge computing[2].

However, the compilation of unsafe languages to Wasm opens up the opportunity for the introduction of new classes of vulnerabilities into the setting of Web programs, as vulnerabilities in the original programs can be transposed to Wasm binaries and resurface in the Web under a new guise [5]. This is the case of buffer overflows [6], format string bugs [7], or use-after-free errors[3]. By exploiting such flaws[4], cyber attackers have access to a widened surface for launching serious attacks on the Web. These include cross-site scripting attacks by exploiting client-side code [6], or code injection attacks by targeting vulnerabilities in server-side code (e.g., powered by Node.js). WebAssembly itself can be used for writing malware, e.g., web keyloggers[5], or crypto-miners [8]. Importantly, Wasm binaries are often integrated directly into Web applications, with developers not having access to the corresponding source code. Hence, developers must analyse standalone Wasm code to test it against potential security vulnerabilities and other types of execution errors.

Symbolic execution 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/intermediate languages. For instance, KLEE [9] found several fatal bugs in high-profile software systems, such as in GNU COREUTILS[6] and BUSYBOX[7]; DART [10] successfully generated inputs to crash 65% of the external functions in oSIP[8], an open-source session initiation protocol; and *Java PathFinder* [11], developed by NASA, was used to test the Orion Spacecraft's control software. Nonetheless, to the best of our knowledge, there are only two tools for symbolically executing Wasm code: WANA [12] and Manticore [13]. These tools are specifically aimed at smart contracts written in Wasm and are also in preliminary development stages. Importantly, the analysis of smart contracts is fundamentally different from that of general-purpose software applications, which are often much larger and, therefore, impose different scale requirements.

We present the WebAssembly Symbolic Processor, WASP, a novel symbolic execution engine for testing Wasm modules, which was built on top of the standard-compliant Wasm reference implementation introduced in [1]. The symbolic execution engine at the core of WASP was obtained by lifting

[1] https://nodejs.org/en/blog/release/v12.3.0/
[2] https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime
[3] https://www.exploit-db.com/exploits/46968
[4] https://www.fastly.com/blog/hijacking-control-flow-webassembly
[5] https://www.virusbulletin.com/virusbulletin/2018/10/dark-side-webassembly/
[6] https://www.gnu.org/software/coreutils/
[7] https://busybox.net/
[8] http://www.gnu.org/software/osip/osip.html

the reference interpreter[9] from concrete values to symbolic values. In order to achieve this, we formalised the symbolic semantics of Wasm and designed our symbolic interpreter closely following the proposed semantic rules. Both our symbolic semantics and symbolic engine follow the so-called concolic discipline [10, 14], combining concrete execution with symbolic execution and exploring one execution path at a time. A significant advantage of this approach compared to the fully static one is that the latter requires less frequent interactions with the underlying first-order solver; essentially, one call to the solver per explored execution path.

Using WASP, we created WASP-C, a new symbolic execution framework for testing C programs. WASP-C takes as input a C program optionally annotated with assumptions and assertions and generates a test suite for that program. A test suite is a set of test cases, each corresponding to a different execution path of the program to be analysed. If, during symbolic execution, WASP-C finds an assertion violation or a runtime error, the error is reported to the developer together with the concrete inputs that triggered it.

While in this work, we use WASP to build a symbolic execution engine for C, WASP can also be used to obtain symbolic engines for other programming languages provided that they compile to Wasm. In a nutshell, if one wants to use WASP to enable a symbolic execution engine for a given language, one has two key tasks to accomplish. First, the symbolic primitives of WASP, such as the declaration of assertions and assumptions and the creation of symbolic variables, must be exposed at the source-language level and be properly connected to the corresponding WASP primitives via compilation. Second, one must guarantee that 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.

WASP-C was thoroughly evaluated:

- It was used to test Collections-C[10] symbolically, a widely-used generic data structure library for C previously tested using the Gillian-C tool [15]. WASP-C found three bugs in Collections-C, including a previously unknown bug that Gillian-C did not detect. Furthermore, WASP-C is more efficient than Gillian-C, completing the symbolic analysis of the library $1.14\times$ faster.

- It was used to symbolically test the Amazon Encryption SDK[11] for C, generating a high-coverage test suite for that library and demonstrating that WASP-C does indeed scale to industry-grade code.

- It was thoroughly tested against the Test-Comp [16] benchmark, obtaining results comparable to well-established symbolic execution and testing tools for C, such as KLEE [9] and VeriFuzz [17]. 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 [2]), we conclude that WASP-C is the

[9] https://github.com/WebAssembly/spec/tree/master/interpreter
[10] https://github.com/srdja/Collections-C
[11] https://github.com/aws/aws-encryption-sdk-c

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 (including WASP).

In summary, the contributions of this work are three-fold: A robust concolic execution engine named WebAssembly symbolic processor (WASP) v1.0 built on the pre-existing WASP v0.1. Then WASP-C, a symbolic execution framework for testing C programs. Lastly, Three sets of symbolic benchmarks in Wasm with varying degrees of complexity, testing different types of symbolic reasoning.

We structure this thesis as follows: Chapter 2, provides an overview of Wasm focusing on its instruction format and semantics, and we briefly explain symbolic execution. Chapter 3 unfolds all the related research relevant to the work we performed in this thesis. In Chapter 4, we start by giving an overview of the original version of WASP (4.1), then we describe the improvements over the symbolic memory (4.2), followed by necessary changes to the first order solver to support this new memory model (4.3). In this chapter, we also introduce a new mechanism that allows WASP to recover from failed assumptions (4.4). In Chapter 5, we present WASP-C; we describe how we implemented the runtime models(5.2) together with a technique for efficient test suite generation (5.3). In Chapter 6, we evaluate WASP-C. Finally, Chapter 7 reflects on our work and points out future research directions.

2

Background

Contents

(value types) $t ::=$ i32 | i64 | f32 | f64
(packed types) $tp ::=$ i8 | i16 | i32
(function types) $tf ::= t^* \to t^*$

(instructions) $e ::=$ **unreachable** | **nop** | **drop** | **select** | t.**const** c |
get_local i | **set_local** i | **tee_local** i | **get_global** i |
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 |
$t.unop_t$ | $t.binop_t$ | $t.testop_t$ | $t.relop_t$ | $t.cvtop\ t\_sx^?$

(functions) $f ::= ex^*$ **func** tf **local** t^* e^* | ex^* **func** tf im
(imports) $im ::=$ **import** "$name$" "$name$"
(exports) $ex ::=$ **export** "$name$"

Figure 2.1: Simplified Wasm abstract syntax, as presented in [1].

This chapter provides the necessary background for the rest of this thesis. In Section 2.1 we will start by giving an overview of Wasm as a language, where we present an example of its execution semantics (2.1.2). In this section, we also briefly explore compiler and toolchain technologies that compile C/C++ programs to Wasm modules (2.1.3). In Section 2.2 we introduce symbolic execution. Finally, we will wrap up the chapter with a summary.

2.1 WebAssembly

2.1.1 Syntax

WebAssembly (abbreviated Wasm), introduced in [1], is low-level bytecode that offers compact representation, efficient validation and compilation, and ensures safe execution with minimal overhead. Since Wasm is an abstraction over common hardware, it makes it language-, hardware- and platform-independent. Like other assembly languages, it is mainly used as a compilation target for 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. Even though Wasm is a binary code format, it has a language with syntax and structure, bringing the opportunity for developers to write this format by hand if desired.

A WebAssembly binary takes the form of a module. A Wasm module can be seen as a collection of Wasm *functions*, together with the declaration of their shared global variables and the specification of the linear memory in which the module is to be executed. 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 handles traps, i.e. unrecoverable Wasm errors, defines how modules are loaded, resolves imports and exports

between modules, and handles I/O and timers. The syntax of Wasm programs is given in Figure 2.1 and includes: functions f, instructions e, values c, value types t, packed types tp, and function types tf.

Wasm Types and Values Wasm has only four primitive types, all of which are readily available on common hardware. Those are integers and IEEE 754 floating-point numbers, each with a 32 and 64-bit variant. Wasm has no distinction between signed and unsigned *integers*; instead, instructions have a *sign extension* to indicate how to interpret the generated integer values.

Wasm Instructions Wasm instructions can be broadly divided into the following categories: **(1)** stack instructions, for explicit manipulation of the stack, **(2)** variable instructions, for updating and retrieving the values of both local and global variables, **(3)** memory instructions, for loading and storing values to and from the module's memory, **(4)** control flow instructions, for determining the control flow of the program, and **(5)** multiple unary and binary operators over Wasm primitive types. The stack instructions are straightforward: the instruction **drop** pops the value at the top of the stack, and the instruction **const** pushes its argument onto the stack. The unary and binary operators include the standard relational, arithmetic, and boolean operators over the Wasm integers or floats. In the following, we discuss the variable, memory and control flow instructions, introducing at the same time the key structures that they interact with.

Variable Instructions 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. Wasm provides instructions to set the value of a given variable to the value at top of the stack (**set_local** and **set_global** for local and global variables respectively) and to get the value of a given variable, pushing it onto the top of the stack (**get_local** and **get_global** for local and global variables respectively). For instance, the instruction **get_local** i is used to push the value of the i^{th} local variable onto the top of the stack and the instruction **set_local** i is used to store the value on top of the stack on the i^{th} local variable. Observe that both instructions **set_global** and **set_local** have the side effect of popping the value at the top of the stack. As it is often convenient to assign a given value to a local variable without popping that value out of the stack, Wasm includes the instruction **tee_local** for doing precisely that. This instruction is equivalent to a **set_local** followed by a **get_local** (e.g. **tee_local** $i \equiv$ **set_local** i; **get_local** i).

Memory Instructions: The primary storage of a Wasm module is a large array of bytes, commonly referred to as *linear memory*. A module can only have one memory, but memories can be exported/imported, meaning modules can share the same memory. The initial memory size is fixed. However,

memories can be dynamically grown using the instruction **grow_memory**. The size of a memory is increased one page at a time – page size is fixed at 64 KiB. Memory is accessed through the **load** and **store** instructions, using packed types (integers with 8, 16 or 32 bits) to define the size of the memory block to be read. For instance, the instruction $i32.$***load8_u*** specifies that an 8-bit integer will be loaded from memory and pushed onto the top of the stack.

Control Flow Instructions In contrast to most stack machines, Wasm presents structured control flow constructs with **loop**, **if** and **block** instructions, ensuring that humans easily interpret the code and that no irreducible loops [18] are encountered, as well as branches to blocks with unaligned stack heights and branches into the middle of a multi-byte instruction.

2.1.2 Semantics

The semantics of Wasm is divided into three phases. The core specification [19] specifies each of them:

1. **Decoding:** Wasm is transmitted in a binary encoding of the abstract syntax presented in Figure 2.1. Decoding is converting the binary into an internal representation, which can then be used by the embedder or, in case of an actual implementation by the machine.
2. **Validation:** Decoded modules are validated through the use of validation rules defined as a type system. A type system is composed of a set of rules that are meant to verify that, at all times, the stack contains values of the correct type. The authors [19] refer that the Wasm type system is *sound*, which implies that well-typed Wasm programs cannot generate a stack whose values are not consistent with their declared types.
3. **Execution:** After decoding and validation, a valid module can be executed. Execution is done as a two-step process:

   (a) **Instantiation:** A module is a static representation of a program; therefore, it must be instantiated. An instance of a module is the computation artefact that supports its runtime execution, complete with its state and execution stack. Instantiation requires definitions for all imports, and it also initialises global variables and memories. The result is an instance for all the module's exports.

   (b) **Invocation:** Wasm computation can be initiated by invoking an exported function of an instance. Exported functions work as the entry points to a Wasm module, akin to the main function of a C program.

Execution Semantics The authors of the Wasm specification [1] also propose a small-step operational semantics for the language. Small-step operational semantics [20] is a mathematical device for rigorously defining the semantics of programming languages. Small-step semantics describe the behaviour of a program one step at a time and are formally defined using a set of rules that are applied to program configurations. Using these rules, a program is interpreted by iteratively applying the appropriate rule to the current configuration until no rule can be further applied.

In the following, we will briefly recap the operational semantics of Wasm following the original semantics and corresponding interpreter[1]. To this end, we must first introduce Wasm semantic domains, which we summarise in the table below. A Wasm configuration is composed of five elements: **(1)** the instruction to be executed, e, **(2)** the local store, $csto$, mapping the local variable indexes to their respective values, **(3)** the stack st, consisting of a list of concrete values following the last-in-first-out (LIFO) discipline; **(4)** the global store δ mapping the global variable indexes to their associated concrete values, and **(5)** the memory μ, consisting of a sequence of bits, which can be accessed through memory addresses.

Wasm Concrete Semantic Domains

CONFIGURATION	C	:	$\langle e,\ \rho,\ st,\ \delta,\ \mu \rangle$
LOCAL STORE	ρ	:	$i32 \rightarrow c$
STACK	st	:	$c\ list$
MEMORY	μ	:	$i32 \rightarrow \{0,1\}^*$
GLOBAL STORE	δ	:	$i32 \rightarrow c$

OUTCOME	$oi \in OI ::=$	$e \mid \cdot \mid \textit{Trap}$
CONCRETE VALUES	$c ::=$	$i32 \mid i64 \mid f32 \mid f64$

Semantic judgements are of the form $e, \rho, st, \delta, \mu \Rightarrow_{\mathsf{c}} \rho', st', \delta', \mu', o$ meaning that the evaluation of the instruction e, along with the current local store ρ, the stack st, the global store δ, and the memory μ, results in a new local store ρ', stack st', global store δ', memory μ', and outcome o. The *outcome* o is used to indicate if the current evaluation leads to the evaluation of another instruction e, or if no other instruction is subsequently evaluated $\cdot$. Outcomes can also take the form of a *Trap*, causing the current computation to abort. Traps are not handled by Wasm but by its embedder, which handles the errors generated by Wasm traps.

A selection of the rules that comprise the relation $\Rightarrow_{\mathsf{c}}$ are given in Figure 2.2 and explained below. Note that in semantic transitions, we omit the elements of the configuration that are neither updated nor inspected by the current rule, writing, for instance, $e, \rho, st \Rightarrow_{\mathsf{c}} \rho', st', o$ to mean $e, \rho, st, \delta, \mu \Rightarrow_{\mathsf{c}} \rho', st', \delta, \mu, o$.

IF The IF rules check the value c at the top of the stack. If $c \neq 0$, the semantics generates the outcome **block** $(\overline{e}_1)$, meaning that the "then" block is to be executed. Otherwise, it generates the outcome **block** $(\overline{e}_2)$, signifying that the "else" block is to be executed. The **block** rule evaluates each instruction step-by-step, while keeping track of the evaluation context.

[1] https://github.com/WebAssembly/spec/tree/master/interpreter

IF-TRUE

$$\frac{c \neq 0}{\textbf{if}\ (\overline{e}_1)(\overline{e}_2), (c :: st) \Rightarrow_{\mathsf{c}} st, \textbf{block}(\overline{e}_1)}$$

IF-FALSE

$$\frac{c = 0}{\textbf{if}\ (\overline{e}_1)(\overline{e}_2), (c :: st) \Rightarrow_{\mathsf{c}} st, \textbf{block}(\overline{e}_2)}$$

STORE

$$\frac{\mu' = \mu[(k+o) \rightarrow bits_t(c)]}{t.\textbf{store}\ o, (k :: c :: st), \mu \Rightarrow_{\mathsf{c}} st, \mu', \cdot}$$

LOAD

$$\frac{v := fromBits_t(\mu(k+o))}{t.\textbf{load}\ o, (k :: st), \mu \Rightarrow_{\mathsf{c}} (v :: st), \mu, \cdot}$$

GETLOCAL

$$\frac{}{\textbf{get\_local}\ i, \rho, st \Rightarrow_{\mathsf{c}} \rho, (\rho(i) :: st), \cdot}$$

SETLOCAL

$$\frac{\rho' = \rho[i \rightarrow v]}{\textbf{set\_local}\ i, \rho, (v :: st) \Rightarrow_{\mathsf{c}} \rho', st, \cdot}$$

$$e, \rho, st, \delta, \mu \Rightarrow_{\mathsf{c}} \rho', st', \delta', \mu', o$$

Figure 2.2: Wasm concrete semantics.

STORE This rule takes the **store** instruction with a parameter o, denoting the memory offset where the store is going to take place, the local store ρ, a stack with the values k and c on top $k :: c :: st$, and the current memory μ. The values on top of the stack correspond to the memory address k, and the concrete value c to be stored in this address. Given this, the memory is updated to map the real address $(k + o)$ to the value c, with $\mu' = \mu[(k + o) \rightarrow \textit{bits}_t(c)]$. The other components of the given configuration are left unchanged.

LOAD This rule takes the **load** instruction with an offset o, the local store ρ, a stack with an address k on top $k :: st$, and a current memory μ, and puts the value $v := \textit{fromBits}_t(\mu(k+o))$ at the real address on top of the stack $v :: st$. Note that the `load` and `store` instructions originally involve an alignment parameter as per Figure 2.1. However, since the extra alignment parameter does not affect the behaviour of these instructions, being only used by the Wasm type system, we omit it from the presented semantics.

GETLOCAL This rule obtains the value associated with local variable index by i from the store ρ, and puts it on top of the stack, generating the new stack $(\rho(i) :: st)$.

SETLOCAL This rule updates the entry i of the local variable store ρ, associating the value v on top of the stack with local variable indexed by i, resulting in the new local store $\rho' = \rho[i \rightarrow v]$ and no resulting instruction.

Implementation The authors of [1] developed a reference interpreter for Wasm in *OCaml*, closely following the proposed operational semantics. This interpreter can: **(1)** parse and validate modules in binary or text format, **(2)** execute scripts with module definitions, invocations, and assertions, **(3)** convert between Wasm's binary and text format, **(4)** export text scripts to self-contained JavaScript test cases and **(5)** run as an interactive interpreter.

```c
int main() {
    int x = 1, y = 0;
    int a = 4, b = 2;
    if (x > 0) {
        b = 6;
        if (x < y)
            a = (x * 2) + y;
    }
    return a != b;
}
```

Listing 2.1: Simple concrete program in C.

Running Example To illustrate the execution semantics of Wasm, we will now show how it can be used to model the execution of the Wasm program given in Listing 2.2. This program results from the compilation of the C program given in Listing 2.1. Importantly, we represent the Wasm program in Wasm Textual Format (WAT), which allows program variables to be named. However, the Wasm interpreter replaces named variables with their corresponding indexes at load time, effectively converting the given Wasm program to the syntax introduced in the previous section. Figure 2.3 shows the concrete execution flow of the given Wasm program, with each node in the flow diagram representing a Wasm instruction. At the left of each node, we represent the state of the stack after the execution of the corresponding instruction. Stacks grow downward, meaning that the top of the stack corresponds to its bottommost cell.

```
(func $main
  ;; initialize x=1, y=0, a=4, b=2...
  (i32.const 0)
  (i32.gt_s)
  (if
    (then
      (i32.const 6)
      (local.set $b)
      (local.get $x)
      (local.get $y)
      (i32.lt_s)
      (if
        (then
          (local.get $x)
          (i32.const 2)
          (i32.mul)
          (local.get $y)
          (i32.add)
          (local.set $a)))))
  (local.get $a)
  (local.get $b)
  (i32.ne))
```

Listing 2.2: Wasm module for the program in Listing 2.1.

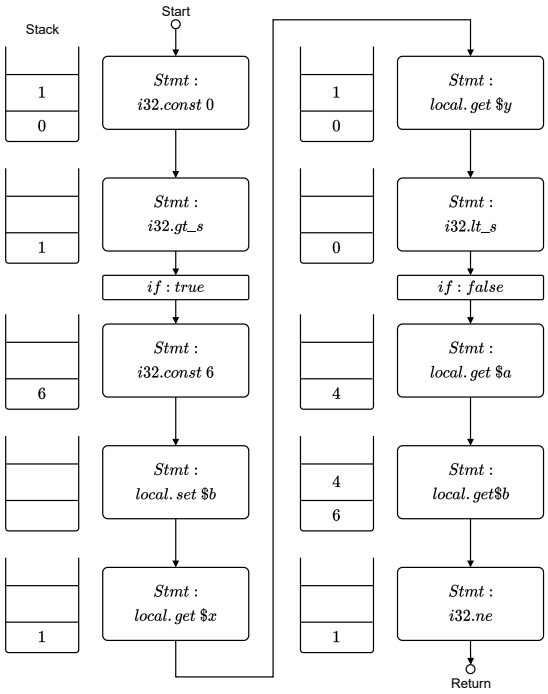


Figure 2.3: Concrete execution flow of the program in Listing 2.1.

The Wasm interpreter executes the given program as follows:

- In line 2, it pushes the value of variable `x` onto the stack. Next, it pushes the constant 0 onto the stack and performs a *greater than* relational operation between the two topmost elements of the stack (i.e., `x > 0`), which are replaced by the integer 1, used to represent the Boolean value true.
- In line 5, the evaluation of the if instruction outputs its "then" branch as the next instruction to be evaluated, given that it finds the integer 1 on top of the stack.
- In lines 7–8, the interpreter sets the value of the local variable `b` to 6.
- In lines 9–11, the interpreter performs the relational operation `x < y`, which given the values in the local store evaluates to the integer 0, used to represent the Boolean value false.
- In line 12, the evaluation of the if instruction generates the empty outcome, given that it finds the value 0 on top of the stack and the given if the instruction has no "else" branch.
- Finally, in lines 20–22, the interpreter performs the relational operation `a != b`, and places its return value on top of the stack.

2.1.3 Compilation

Assembly languages are low-level programming languages with a strong connection between the instructions in the language and the machine code instructions for some architecture. However, Wasm is an abstraction over modern hardware, making it language-, hardware-, and platform-independent. Consequently, Wasm instructions are called virtual instructions, and Wasm is considered a *Virtual Instruction Set Architecture* (V-ISA).

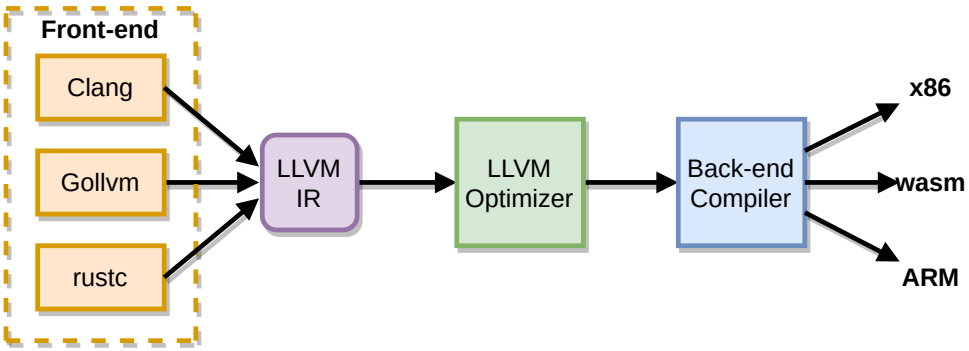


Figure 2.4: LLVM Compiler Toolchain Pipeline.

Compiler Toolchain Like other assembly languages, Wasm is a compilation target for high-level programming languages, such as C/C++ or Rust. Currently, the available set of tools that allows us to

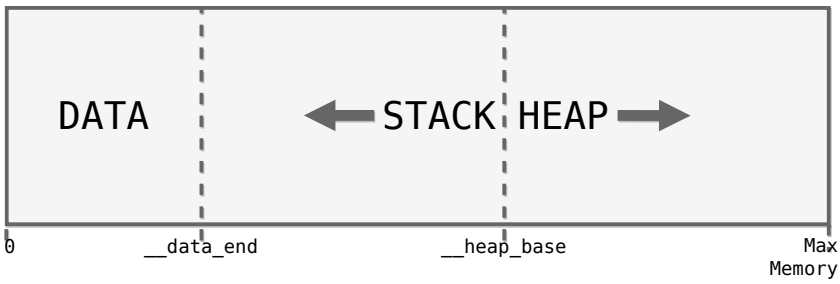


Figure 2.5: LLVM's memory layout for Wasm modules.

compile a program, for example, in the C language to Wasm, are provided by the *LLVM Project*[2]. The LLVM project is a collection of modular and reusable compiler and toolchain technologies. It is modular because it has a front-end compiler that compiles code into an intermediate representation (IR), also called LLVM. Then a back-end compiler translates the IR into its target, as exemplified in Figure 2.4. Wasm is just one of the many targets the LLVM Project supports. For instance, for the C language, there exists a front-end compiler called Clang[3] that can emit byte code in LLVM IR. With byte code in the LLVM IR, LLVM can perform various optimisations and eventually, with the appropriate back-end compiler, it can translate the optimised byte code into a binary Wasm module.

Emscripten [21] is a compiler toolchain for C/C++ initially targeting JavaScript, but which now also supports Wasm. It uses Clang and LLVM to compile to Wasm, through the same process as that given in Figure 2.4. Code generated from Emscripten is meant to run in a JavaScript engine, typically on a web browser. This has implications for the kind of runtime environment that can be generated. For example, on the web, there is no direct access to the local file system. For this reason, Emscripten comes with a partial implementation of a C library, mostly written from scratch in JavaScript with parts compiled from an existent C library [21].

Emscripten has been applied successfully on several real-world code bases. It was used to compile the Python C[4] and Lua C[5] implementations, allowing Python and Lua code to run on the web.

Compiling C to Wasm Let us now take a closer look at how C programs are compiled to Wasm. LLVM's Wasm object linker dictates the way the C memory is represented in Wasm, `wasm-ld`[6]. Figure 2.5 illustrates how the C heap and stack are represented in the Wasm linear memory. We can see that the stack grows downwards into lower linear memory addresses, while the heap grows upwards into higher linear memory addresses. More concretely, `__heap_base` is a pointer to the start of the heap segment of the linear memory, and `__data_end` is a pointer to the start of the stack segment of the linear

[2] http://llvm.org/
[3] http://clang.llvm.org
[4] https://github.com/iodide-project/pyodide
[5] https://daurnimator.github.io/lua.vm.js/lua.vm.js.html
[6] https://lld.llvm.org/WebAssembly.html

```
;; Compiled from add.c:
;;     int add (int a, int b) { return a + b; }
(module
  (type (;0;) (func))
  (type (;1;) (func (param i32 i32) (result i32)))
  (func $add (type 1) (param i32 i32) (result i32)
    local.get 1
    local.get 0
    i32.add)
  (memory (;0;) 2)
  (global (;0;) (mut i32) (i32.const 66560))
  (global (;2;) i32 (i32.const 1024))
  (global (;3;) i32 (i32.const 1024))
  (global (;4;) i32 (i32.const 66560))
  (export "memory" (memory 0))
  (export "add" (func $add))
  (export "__data_end" (global 2))
  (export "__global_base" (global 3))
  (export "__heap_base" (global 4))
```

Listing 2.3: Example if a compiled Wasm module in its textual representation.

memory. The first segment of memory corresponds to the data segment of the compiled C program; recall that the data segment is the region of memory used to store the initialised static variables of the program. Given this memory layout, the stack segment of memory can only grow to a maximum of `__heap_base - __data_end`. In contrast, the heap segment can grow dynamically as need be using the instruction **grow_memory**, which allows the program to grow the linear memory by extending it by one memory page, i.e. an array of 64KiB.

Anatomy of Wasm Modules Listing 2.3 demonstrates the textual representation of a binary module compiled from a C program. A binary is divided into sections according to the different kinds of entities declared in it [19]. In this example, we have five sections. First we have a *type section*, which is a collection of all function types. Next, the *function section*, which contains function definitions. Next, we have a *memory section*, which specifies the initial size of the memory, set to two pages of 64KiB in this example. Lastly, we have the *global section*, which contains four definitions of global variables and the *export section*, which declares the functions, memories and variables of the module that can be externally accessed.

2.2 Symbolic Execution

Symbolic execution is a program analysis technique used to explore all feasible paths of a program up to a bound. 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 in order 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 Satisfiability Modulo Theories (SMT) solver to check the feasibility of execution paths and check 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 a developer-supplied assertion holds at a given program point if the path condition implies it.

Concolic Execution Concolic execution is a special variation of symbolic execution, in which one pairs up a concrete execution with a purely symbolic execution to avoid interactions with the underlying SMT solver by exploring one execution path at a time. 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 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 solver one time per explored path. Note that in purely symbolic execution, the engine must query the solver every time it hits a branching point in order to determine whether or not its then- and else- branches are feasible.

```
1  int main() {
2    int x = symbolic(), y = symbolic();
3    int a = 4, b = 2;
4    if (x > 0) {
5      b = a + 2;
6      if (x < y)
7        a = (x * 2) + y;
8    }
9    assert(a != b);
10 }
```

Listing 2.4: Concolic execution example in C, adapted from Listing 2.1.

Concolic Execution: Example Let us now take a look at how concolic execution works in practice. Consider the program given in Listing 2.4. 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 this assertion always holds, one has to explore all the possible execution paths of the program, which we illustrate in Figure 2.6 in the form of an execution tree. As shown in the figure, the assertion does not hold when $x = 1$ and $y = 4$, which cause variables a and b to be assigned to 6, violating the final assert statement. Below, we will show how these inputs can be found using concolic execution.

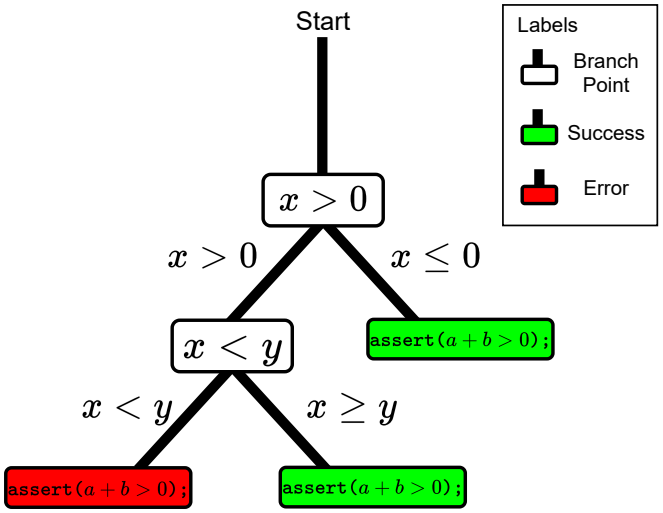


Figure 2.6: Paths explored during concolic execution of the program in Listing 2.4.

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 type. 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: $x \leq 0$.

Before the second concolic iteration, the concolic execution engine queries the underlying SMT 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 concolic execution engine queries the solver for a model for the symbolic inputs that satisfy 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.

When the concolic symbolic engine finds the assert, the concolic state is as follows:

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

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 SMT 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.

Summary

This chapter provides the necessary background for the rest of this thesis. First, we introduced Wasm, reviewing its syntactic structure, semantic domains, and concrete semantics. Then, we introduced the standard technologies for compiling high-level languages, such as C/C++ or Rust to Wasm, briefly explaining how the C memory is represented at the Wasm level. We finish with a succinct account of symbolic and concolic execution, illustrating how concolic execution works using a simple example. The following chapter overviews the relevant research work on these subjects.

3

Related Work

Contents

In this chapter we discuss the related work. We first discuss both *classic symbolic execution* and *concolic execution*, giving an overview of state-of-the-art tools that use these kinds of analyses. Secondly, we briefly look into modelling runtime system-level interactions in the context of symbolic execution. Thirdly, we review techniques for implementing symbolic memory models in the context of symbolic execution tools. Lastly, we discuss state-of-the-art analysis tools for Wasm.

3.1 Symbolic Execution: Overview

Symbolic execution has historically been used to discover severe errors in a vast spectrum of systems, such as web servers [22], file systems [23], and device drivers [24]. Additionally it has been successfully applied to a wide range of programming languages such as C [10], Java [14], and Python [25]. In the context of the web, there are several state-of-the-art tools to symbolically execute JavaScript programs [26–30], which confirms the relevance of symbolic execution for the analysis and testing of web applications.

Symbolic execution is divided into two main classes: static and dynamic (also referred to as concolic). Static symbolic execution engines, such as [26, 27, 31–34] explore the entire symbolic execution tree up to a pre-established depth. Concolic execution engines, such as [9, 10, 14, 28–30, 35], work by pairing up symbolic execution and concrete execution so that the symbolic execution may fall back to the concrete execution whenever it produces symbolic formulas that are not supported by the underlying first order solver. As consequence, a major advantage of concolic execution over classic symbolic execution is that concolic execution requires less interactions with the underlying solver and a simpler memory model.

Given the limited scope of this thesis and the dimension of the body of research on both static and dynamic symbolic execution tools, we encourage the reader to see [36–38] for more comprehensive surveys on the topic. We continue by briefly surveying related work regarding static symbolic execution and then exploring more deeply the related work regarding concolic execution, as this is the main focus of our work.

3.1.1 Static Symbolic Execution

Static symbolic execution was the pioneer technique to symbolically analyse programs. Table 3.1 summarises some state-of-the-art symbolic execution tools that apply this classic technique.

In a nutshell, the tools using the classic symbolic execution can theoretically enumerate all possible control flow paths that are possible in a program up to a bound. Thus, modelling all these runs would provide a very sound analysis of the software being verified. However, this task is quite unfeasible, especially on big software projects, requiring the tools to make compromises for the sake of efficiency.

Tool	Languages	Applications
EFFIGY [31]	PL/I	Program analysis, Interactive debugging
Symbolic PathFinder [32]	Java	Directed incremental symbolic execution [39], Java PathFinder [11] continuity, Probabilistic symbolic execution [40]
Generalized framework [33]	Java	Correctness checking of distributed algorithms; Test input generation for flight software
Rosette [34]	Racket	Implementation of solver-aided languages.
Cosette [26]	JavaScript	Whole program symbolic testing of JavaScript libraries; Compositional debugging of separation logic specifications of JavaScript programs
JaVerT 2.0 [27]	JavaScript	Whole program symbolic testing, verification and automatic compositional testing
KLEE [9]	C, C++, Rust	Automatic generation of high-coverage tests on environmentally-intensive programs

Table 3.1: Representative tools implementing classic symbolic execution.

Tool	Languages	Applications
DART [10]	C	Session Initiation Protocol[1]
CUTE [14]	C	Scalable concolic execution for code with dynamic data structures
jCUTE [14]	Java	Scalable concolic execution for code with dynamic data structures
MAYHEM [35]	x86 Assembly	Find exploitable vulnerabilities in Windows and Linux programs
Kudzu [28]	JavaScript	Disclosure of client-side code injection vulnerabilities
SymJS [29]	JavaScript	Automatic testing of client-side JavaScript web applications
MultiSE [30]	JavaScript	Multi-path symbolic execution

Table 3.2: Representative tools implementing some form of concolic execution.

3.1.2 Concolic Execution

Concolic execution uses a combination of symbolic and concrete execution in order to mitigate the issues inherent to static symbolic execution. Table 3.2 presents symbolic execution engines that employ *concolic* techniques.

DART *DART* [10] provides means to automatically test C programs. It detects errors such as program crashes, violated assertions and non-termination. DART tests each function in isolation and does not consider preconditions. Concretely, it consists of three parts: directed generation of test inputs, automated extraction of unit interfaces from source code, and random generation of test inputs.

CUTE *CUTE* [14] is a concolic unit testing engine for programs written in C. Unlike *DART*, it targets functions with preconditions such as data structure implementations. Moreover, it also achieves better performance than DART by optimising the underlying constraint solver, specialising it for the path conditions that arise in *concolic* execution.

MAYHEM *MAYHEM* [35] is a concolic execution engine for Windows and ELF binary programs. It relies on partial memory modelling to reason about memory at the binary level.

Kudzu *Kudzu* [28] is a system for concolically executing JavaScript Web applications. It relies on black box testing, given the URL of the web application, it generates test cases to explore the execution space of the application. *Kuduzu* was able to successfully find code injection vulnerabilities in client-side web applications. It was also reported that it does not produce false positives.

SymJS *SymJS* [29] is another framework for automatic testing of client-side JavaScript applications. In contrast to *Kuduzu*, which mainly focuses on string reasoning, *SymJS* synthesises sequences of Web events in order to dynamically explore event-driven JavaScript code.

MultiSE *MultiSE* [30], as the name suggests, performs multi-path symbolic execution of JavaScript programs. To reduce the number of symbolic paths, *MultiSE* employs a algorithm for incrementally merging symbolic states together during symbolic execution. However, this techniques requires a different representation of concolic states, where variables are mapped to a set of symbolic expressions.

3.2 Symbolic Execution: Runtime Models

An essential aspect of a symbolic execution engine is handling a program's interaction with the surrounding software stack. A typical example is the data flows that take place through the underlying operating system, such as reading data from a file, receiving network packets through a sockets, and reading/setting environment variables. Therefore, the degree to which the engine can detect these symbolic flows is directly related to its soundness.

In [36, 38] we have a comprehensive overview of the various methods state-of-the-art tools used to deal with this problem. In tools such as DART [10], and CUTE [14], system interactions are dealt with by invoking the system-level function with concrete arguments. The major downside of this approach is the inability to explore the entire state space of the program.

The other approach to handle system-level calls is to create summaries of their side effects [9,41,42]. These summaries essentially model the behaviour of the function code that is being called. In KLEE [9], we have a detailed example of this technique utilised to interact with the file system. A call to `fopen()`

needs to return either a valid file descriptor or some error code. Thus, if the call is made with a symbolic argument, then KLEE will refer to its *symbolic file system*, which consists of a directory with N symbolic files whose number and sizes are specified by the user. Consequently, an operation on a symbolic file is a branch point with $N+1$ possible outcomes, N returns a file descriptor to one symbolic file and one that fails. The downside of this approach is that typically the user must generate these summaries manually.

3.3 Symbolic Execution: Memory Models

Another critical aspect of symbolic execution is how concrete and symbolic memory should be modelled to support programs with pointers and arrays. This requires extending our notion of a memory store by mapping variables and memory addresses to symbolic expressions of concrete values. In [36], the authors provide a review of work in this area and in [43] on the semantic modelling of storage operations.

In [44], the authors roughly classify memory models as either *high-level* or *low-level*. According to the authors, high-level memory models are models where the model itself provides some guarantees of separation or enforcement of memory bounds. In contrast, low-level models are essentially just arrays of bytes where guarantees, such as those inherent in high-level memory models, are not present and must be enforced through logical assertions.

In [45–47], the authors present some examples of high-level memory modelling. The classic example is the encoding of the "struct" construct in C, where each field is a separate memory store mapping memory addresses to contents. This representation greatly facilitates reasoning over programs that manipulate linked data structures, e.g., linked lists, because when assigning a value to a record in C (`node->next = nnode`), we guarantee the rest of the values that `node` points to are unchanged.

Low-level memory modelling examples are explained in [48, 49]. With this model, a memory store is essentially just a total function, and mapping memory addresses to bytes. In low-level memory models, allocations, loads, and stores must be defined in terms of byte manipulations. Consequently, reasoning about programs and programs transformations is inherently more difficult.

In [44] the authors present CompCert's memory model, which is described as a hybrid memory model. The rationale for this categorisation is that it assures specific properties of a high-level model, such as separation between blocks obtained in memory allocation, bounds checks during memory stores, and a particular return type for loads overlapping a prior store operation. However, it offers no separation over accesses performed within the same memory block, which violates the *principle of spatial separation* [45], requiring additional reasoning over field offsets, much like a low-level model. However, in version 2.0 of CompCert's memory model [3], the authors overcome these limitations by exposing the byte-level in-memory representation of integers and floats and introducing fine-grained, byte-level permissions instead of for memory bounds.

Tool	Analysis Type	Applications
Manticore [13]	Symbolic Execution	Symbolically executes binaries and smart contracts
WANA [12]	Symbolic Execution	Finds vulnerabilities in EOSIO and Ethereum smart contracts in Wasm bytecode format
Octopus [51]	Static analysis framework	Performs control flow analysis, call flow analysis, and disassembles Wasm modules.
EOSafe [52]	Symbolic Execution	Detects vulnerabilities in EOSIO smart contracts in Wasm bytecode format.
Vivienne [53]	Relational Symbolic Execution	Analysis of Wasm cryptographic libraries for constant-time violations.

Table 3.3: Representative for Wasm code analysis.

In Gillian-C [15], the authors provide an OCaml implementation of the concrete memory model of CompCert [3] and another OCaml implementation of a symbolic memory model inspired from CompCertS [50]. In both models, the memory is a collection of separated blocks where each block is an array of a given size. Additionally, pointers are modelled as block-offset pairs: meaning that a pair $[b, o]$ is a pointer to the o-th element in block b. Furthermore, the authors implement permission tables for both memory models, which are said to describe the allowed operations for a given cell (e.g. *Readable* and *Writable*).

3.4 Analysis Tools for Wasm

Now we discuss the current state-of-the-art analysis tools available for Wasm. Table 3.3 summarises some state-of-the-art for static analysis of Wasm code.

Manticore *Manticore* [13] is a symbolic execution framework for binaries and smart contracts. As a result of its modularity, *Manticore* supports traditional computing environments (x86/64, ARM) and exotic ones, such as the EVM. Manticore's modular design consists of four important modules. First, the *core engine* is a generic platform-agnostic symbolic execution engine. Secondly, the *native execution model* abstracts hardware execution to implement the high-level execution interfaces that the core engine expects. The third is the *Ethereum execution model*, which can simulate the entire EVM. Lastly, the *auxiliary modules interface* allows Manticore to use different SMT solvers seamlessly.

Although *Manticore* can simulate the entire EVM because of its Ethereum execution model, it must use *symbolic transactions*, where both value and data are symbolic in order to explore the state space of a contract generically. With the release of version 0.3.3, Manticore is now able to execute Wasm

binaries[2] symbolically. Current development efforts are focused on furthering the support of Wasm, but in the future, developers aim to fully support Ethereum-flavored Wasm[3] that will replace the EVM language.

WANA *WANA* [12] is a symbolic execution engine for Wasm bytecode. WANA supports vulnerability detection of both EOSIO and Ethereum smart contracts. WANA's analysis is composed of three stages. First, it must parse Wasm bytecode. Secondly, WANA traverses the paths of the Wasm code with symbolic inputs. WANA prepares a frame as its execution context, including symbolic arguments, local variables, return values and references to its module, and then sequentially executes the instructions in the function body. Lastly, WANA performs vulnerability analysis. If a vulnerability is detected, then a report is generated.

Octopus *Octopus* [51] is a security analysis framework for Wasm modules and Blockchain Smart Contracts. It currently performs control flow analysis, call flow analysis, disassembles and converts to SSA IR Wasm modules. However, symbolic execution will soon be supported.

EOSafe *EOSafe* [52] is a static analysis framework that can automatically detect vulnerabilities in EOSIO smart contracts in Wasm bytecode. EOSafe leverages Octopus [51] to create a Control Flow Graph (CFG) with disassembled Wasm instructions which are fed into the two-step analysis pipeline. First, using the generated CFG, EOSafe locates suspicious functions. Then, with the symbolic execution engine and a EOSIO library emulator, it performs symbolic execution over the disassembled Wasm instructions to find vulnerabilities.

Vivienne *Vivienne* [53] is an open-source tool to analyse Wasm cryptographic libraries for constant-time violations automatically. Vivienne takes three inputs: **(1)** a Wasm module to analyse, **(2)** a *Security Policy*. i.e., memory regions and input parameters of the entry functions, and **(3)** the entry point, that is the first function to analyse. Then Vivienne performs relational symbolic execution (RelSE) over the Wasm code on the entry function, reporting any constant-time violation.

Summary

First, we saw that tools for the verification of web application exist in the context of JavaScript programs [26–30]. Then, we learned there exist tools for symbolically analysing generic Wasm modules, such as Manticore [13] and WANA [12]. However, all of these tools use classic symbolic execution in

[2] https://blog.trailofbits.com/2020/01/31/symbolically-executing-webassembly-in-manticore/
[3] https://github.com/ewasm/design

order to perform their analysis, and as we have seen in [26, 27, 31–34], this type of analysis has major limitations regarding performance. With our work, we intend to take the benefits of concolic symbolic execution tools, such as [9, 10, 14, 28–30, 35], and create a robust and efficient tool for the verification of Wasm code, which is gaining territory in the land of Web programming.

4

WASP

Contents

This chapter presents our current concolic execution engine for Wasm named WebAssembly Symbolic Processor (WASP). WASP is written in OCaml on top of the official Wasm reference interpreter[1]. We begin by giving an overview of the design of WASP (Section 4.1). Next, we explain the Wasm symbolic memory model at the core of WASP (Section 4.2) and describe how WASP interacts with its underlying constraint solver (Section 4.3). Lastly, we present an optimisation to concolic execution developed in the context of WASP (Section 4.4).

4.1 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 the 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[1], which we extended with symbolic facilities according to the high-level architecture described in Figure 4.1. 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 constraint solver, Z3 [54].

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 our small-step concolic semantics of Wasm. Concolic execution requires the interpreter to keep track of both the program's concrete and symbolic state. 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 to Z3 formulas using the Z3 OCaml bindings.

[1] https://github.com/WebAssembly/spec/tree/master/interpreter

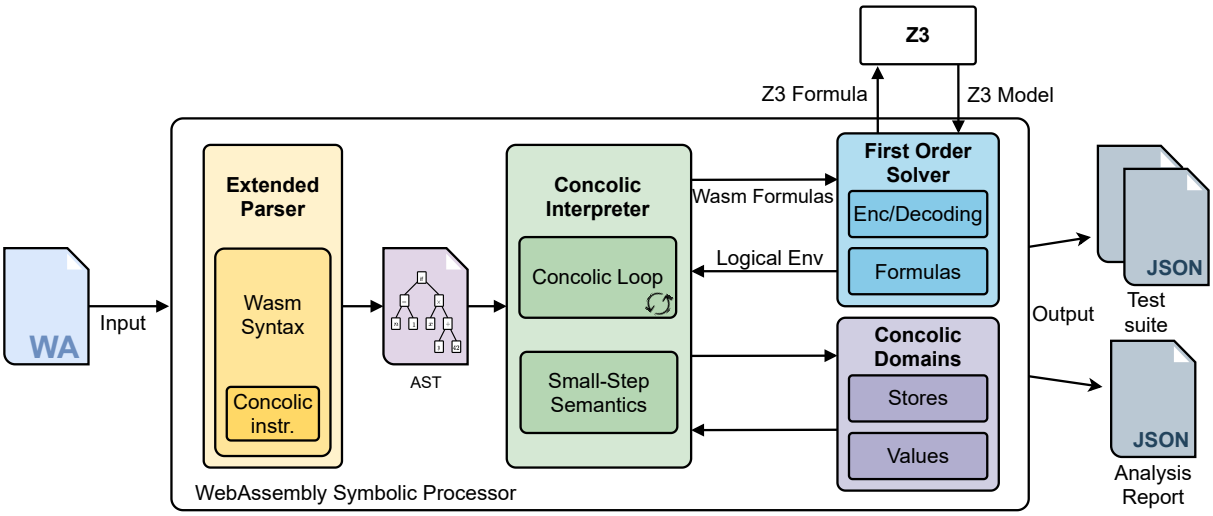


Figure 4.1: High-level architecture of WASP.

In summary, the main components of WASP are the following:

- Extended Parser: an extension of the official Wasm parser that supports the standard instructions for creating symbolic inputs, declaring constraints over those inputs, and checking the constraints that the generated symbolic outputs are supposed to satisfy. Besides these instructions, WASP also supports a variety of instructions for explicitly creating and manipulating first order formulas.

- Concolic Interpreter: an implementation of a concolic interpreter for Wasm that explores all the execution paths of the given program up to a pre-established depth. The concolic interpreter combines concrete and symbolic executions, exploring one execution path at a time and generating for each path its corresponding path condition. Concolic iterations are driven by an evaluation function that follows our small-step concolic semantics of Wasm. Starting from the initial state, our concolic evaluation function iteratively applies the appropriate small-step rule until no rule can be further applied. At the end of each concolic iteration, the concolic interpreter interacts with Z3 to obtain the concrete values of the symbolic inputs for the next concolic iteration. If no such values can be found, the execution terminates.

- Concolic Domains: custom-made data structures representing Wasm's concolic values, stacks and memories. Each concolic domain combines its original concrete version with a custom-made symbolic counterpart. For instance, while concrete stacks are modelled as lists of concrete values, concolic stacks are modelled as lists of concrete values paired with symbolic values. The same applies to Wasm's linear memory. While the concrete linear memory is simply an array of concrete bytes, the concolic linear memory is an array of pairs, each consisting of a concrete and a symbolic byte.

- First Order Solver: an implementation of an encoding from the logic of WASP into the logic of Z3. Every time the concolic interpreter needs to check a given formula's satisfiability, it first converts it into a native Z3 formula and then queries Z3 for the satisfiability of the obtained formula. Our encoding represents Wasm integers and floats, respectively, as Z3 bit-vectors and floats. Observe that the semantics of Wasm's machine integers coincides with that of Z3 bit-vectors; hence, operations on Wasm's integers can be straightforwardly translated to equivalent bit-vector operations at the Z3 level.

WASP 0.1 The version of WASP presented in this work started from a preliminary prototype developed in the context of a previous thesis [55], WASP 0.1. Our version of WASP significantly improves on the original prototype in several respects:

- It includes a new symbolic memory model for Wasm closer to its concrete memory model, allowing for symbolic reasoning about byte-level operations.
- It includes a new encoding of the logic of WASP into that of Z3 based on bit-vectors and floating-point arithmetic in contrast to the original prototype, which used integer and real arithmetic.
- It supports the full syntax of Wasm, while the original prototype only supported a small fragment.
- It was designed and optimised to allow for the execution of real-world C code compiled to Wasm, while the original project was only applied to toy examples directly written in Wasm.

4.1.1 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 concolic semantics operates on concolic states, which can be viewed as pairs of concrete states and symbolic states. Concolic states are therefore inhabited by both concrete values and symbolic values. Formally, symbolic values are given by the following grammar:

$$\hat{s} := c \mid \hat{x} \mid \ominus(\hat{s}) \mid \oplus(\hat{s}, \hat{s}) \mid \otimes(\hat{s}, \hat{s}, \hat{s}) \tag{4.1}$$

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 denoting symbolic byte expressions. In WASP, symbolic variables are prefixed with a '#'.

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:

(instructions) $e ::= \ldots \mid$ **sym_assume** | **sym_assert** | t.**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 or not 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 $\tilde{\mu}$, mapping integer addresses to pairs of concrete bytes and symbolic bytes; **(2)** a concolic local store $\tilde{\rho}$, mapping local variable indexes to pairs of concrete and symbolic values (e.g. $\tilde{\rho} = [0 \mapsto (2, \#y)]$); **(3)** a concolic global store $\tilde{\delta}$ mapping global variable indexes to pairs of concrete and symbolic values (e.g. $\tilde{\delta} = [0 \mapsto (2, \#y)]$); **(4)** a concolic stack $\tilde{st}$, consisting of a sequence of pairs of concrete and symbolic values (e.g. $\tilde{st} = (2, \#y) :: (0, \#x)$); **(5)** a logical environment ε mapping symbolic variables to concrete values (e.g. $\varepsilon = [\#x \mapsto 0, \#y \mapsto 2]$); 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 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, logical environments do not have a counterpart in concrete execution. The concolic interpreter uses the logical environment to keep track of the concrete values associated with the symbolic variables. Essentially, the logical environment stores the bindings of the symbolic variables computed at the beginning of each concolic iteration.

The concolic semantics makes use of computation outcomes [15] to capture the flow of execution. We consider four 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 semantic domains are summarised below. For clarity, we distinguish the concolic domains from their concrete counterparts by adding a tilde over the original symbol; for instance, we use ρ for the concrete local store and $\tilde{\rho}$ for its concolic counterpart.

Concolic Semantic Domains

LOCAL STORE	$\tilde{\rho}$	:	$i32 \rightarrow c \times \hat{s}$
STACK	$\tilde{st}$	:	$(c \times \hat{s})\ list$
LOGICAL ENV	ε	:	$string \rightarrow c$
PATH COND	$\pi \in \Pi$	:	$bool\ \ symbol\ \ expr$
GLOBAL STORE	$\tilde{\delta}$	:	$i32 \rightarrow c \times \hat{s}$
MEMORY	$\tilde{\mu}$	:	$i32 \rightarrow c \times \hat{s}$

OUTCOME	$\tilde{o} ::=$	$e \mid \cdot \mid$ *Trap* $\mid$ *AsrtFail* $\mid$ *AsmFail*
SYMBOLIC EXPR	$\hat{s} ::=$	$c \mid \hat{x} \mid \ominus(\hat{s}) \mid \oplus(\hat{s}, \hat{s})$

IF-TRUE
$$\frac{\tilde{st} = ((c, \hat{s}) :: \tilde{st}') \qquad c \neq 0 \qquad \pi' = ((\hat{s} \neq 0) \wedge \pi)}{\textbf{if}\ (\overline{e}_1)(\overline{e}_2), \tilde{st}, \pi\ \Rightarrow_{\mathsf{cs}}\ \tilde{st}', \pi', \textbf{block}(\overline{e}_1)}$$

IF-FALSE
$$\frac{\tilde{st} = ((c, \hat{s}) :: \tilde{st}') \qquad c = 0 \qquad \pi' = ((\hat{s} = 0) \wedge \pi)}{\textbf{if}\ (\overline{e}_1)(\overline{e}_2), \tilde{st}, \pi\ \Rightarrow_{\mathsf{cs}}\ \tilde{st}', \pi', \textbf{block}(\overline{e}_2)}$$

STORE
$$\frac{\tilde{\mu}' = store\_bytes(\tilde{\mu}, k + o, (c, \hat{s}))}{t.\textbf{store}\ o, ((k, \hat{s}_k) :: (c, \hat{s}) :: \tilde{st}), \tilde{\mu}\ \Rightarrow_{\mathsf{cs}}\ \tilde{st}, \tilde{\mu}', \cdot}$$

LOAD
$$\frac{n = size(t) \qquad (c, \hat{s}') = load\_bytes(\tilde{\mu}, k + o, n)}{t.\textbf{load}\ o, ((k, \hat{s}) :: \tilde{st}), \tilde{\mu}\ \Rightarrow_{\mathsf{cs}}\ ((c, \hat{s}') :: \tilde{st}), \tilde{\mu}, \cdot}$$

GETLOCAL
$$\frac{}{\textbf{get\_local}\ i, \tilde{\rho}, \tilde{st}\ \Rightarrow_{\mathsf{cs}}\ \tilde{\rho}, (\tilde{\rho}(i) :: \tilde{st}), \cdot}$$

SETLOCAL
$$\frac{\rho' = \rho[i \mapsto (c, \hat{s})]}{\textbf{set\_local}\ i, \tilde{\rho}, ((c, \hat{s}) :: \tilde{st})\ \Rightarrow_{\mathsf{cs}}\ \tilde{\rho}', \tilde{st}, \cdot}$$

SYMASSERT
$$\frac{c = 0}{\textbf{sym\_assert}, \tilde{\rho}, ((c, \hat{s}) :: \tilde{st}), \varepsilon, \pi\ \Rightarrow_{\mathsf{cs}}\ \mathsf{AsrtFail}}$$

SYMASSERT
$$\frac{c \neq 0 \qquad (\pi \wedge \neg\hat{s})\ \mathsf{SAT}}{\textbf{sym\_assert}, \tilde{\rho}, ((c, \hat{s}) :: \tilde{st}), \varepsilon, \pi\ \Rightarrow_{\mathsf{cs}}\ \mathsf{AsrtFail}}$$

SYMASSERT
$$\frac{c \neq 0 \qquad (\pi \wedge \neg\hat{s})\ \mathsf{UNSAT}}{\textbf{sym\_assert}, \tilde{\rho}, ((c, \hat{s}) :: \tilde{st}), \varepsilon, \pi\ \Rightarrow_{\mathsf{cs}}\ \tilde{\rho}, \tilde{st}, \varepsilon, \pi, \cdot}$$

SYMASSUME
$$\frac{\tilde{st} = ((c, \hat{s}) :: \tilde{st}') \qquad c = 0 \qquad \pi' = (\neg\hat{s} \wedge \pi)}{\textbf{sym\_assume}, \tilde{\rho}, \tilde{st}, \varepsilon, \pi\ \Rightarrow_{\mathsf{cs}}\ \tilde{\rho}, \tilde{st}', \varepsilon, \pi', \mathsf{AsmFail}}$$

SYMASSUME
$$\frac{\tilde{st} = ((c, \hat{s}) :: \tilde{st}) \qquad c \neq 0 \qquad \pi' = (\hat{s} \wedge \pi)}{\textbf{sym\_assume}, \tilde{\rho}, \tilde{st}, \varepsilon, \pi\ \Rightarrow_{\mathsf{cs}}\ \tilde{\rho}, \tilde{st}', \varepsilon, \pi', \cdot}$$

SYMBOLIC-FRESH
$$\frac{\hat{x} \notin \varepsilon \qquad i \in t \qquad \varepsilon' = \varepsilon[\hat{x} \mapsto i]}{t.\textbf{symbolic}\ \hat{x}, \tilde{st}, \varepsilon\ \Rightarrow_{\mathsf{cs}}\ ((i, \hat{x}) :: \tilde{st}), \varepsilon', \cdot}$$

SYMBOLIC
$$\frac{\hat{x} \in \varepsilon}{t.\textbf{symbolic}\ \hat{x}, \tilde{st}, \varepsilon\ \Rightarrow_{\mathsf{cs}}\ ((\varepsilon(\hat{x}), \hat{x}) :: \tilde{st}), \varepsilon, \cdot}$$

Figure 4.2: WebAssembly symbolic semantics: $e, \tilde{\rho}, \tilde{st}, \varepsilon, \pi, \tilde{\delta}, \tilde{\mu}\ \Rightarrow_{\mathsf{cs}}\ \tilde{\rho}', \tilde{st}', \varepsilon', \pi', \tilde{\delta}', \tilde{\mu}', \tilde{o}$.

We formalise the concolic semantics of Wasm instructions using the mathematical relation $\Rightarrow_{\mathsf{cs}}$. Semantic judgements are of the form:

$$e, \tilde{\rho}, \tilde{st}, \varepsilon, \pi, \tilde{\delta}, \tilde{\mu}\ \Rightarrow_{\mathsf{cs}}\ \tilde{\rho}', \tilde{st}', \varepsilon', \pi', \tilde{\delta}', \tilde{\mu}', \tilde{o} \tag{4.2}$$

meaning that the concolic evaluation of the instruction e in the local store $\tilde{\rho}$, stack $\tilde{st}$, logical environment ε, current path condition π, global store $\tilde{\delta}$, and memory $\tilde{\mu}$ results in a new local store $\tilde{\rho}'$, stack $\tilde{st}'$, logical environment ε', path condition π', global store $\tilde{\delta}'$, memory $\tilde{\mu}'$, and an outcome $\tilde{o}$. This relation is defined by a set of semantic rules of which a representative selection is shown in Figure 4.2 and explained below. Observe that, unlike the concrete semantic rules, the concolic semantic rules are non-deterministic, meaning that more than one rule may be applied at a given point during the execution. As for the concrete semantics, we omit the elements of the configuration that are neither updated nor inspected by the current rule, writing, for instance, $e, \tilde{\rho}, \tilde{st}\ \Rightarrow_{\mathsf{cs}}\ \tilde{\rho}', \tilde{st}', \tilde{o}$ to mean $e, \tilde{\rho}, \tilde{st}, \varepsilon, \pi, \tilde{\delta}, \tilde{\mu}\ \Rightarrow_{\mathsf{cs}}\ \tilde{\rho}', \tilde{st}', \varepsilon, \pi, \tilde{\delta}, \tilde{\mu}, \tilde{o}$. The concolic rules are explained below.

IF The IF rule analyses the concrete value c on top of the stack $(c, \hat{s}) :: \tilde{st}$. If $c \neq 0$, then the path condition is conjoined with the symbolic expression associated with the value on top of the stack $(\hat{s} \neq 0) \wedge \pi$. The resulting outcome is a **block** with the set of instructions $\overline{e}_1$, corresponding to the "then"

branch. If $c = 0$, the opposite happens, the resulting path condition is $(\hat{s} = 0) \wedge \pi$, and the outcome is a block with the set of instructions $\overline{e}_2$, corresponding to the "else" branch.

STORE The STORE rule is very similar to the corresponding rule in the concrete semantics. The difference is that a function called $store\_bytes$ unpacks each byte of $(c, \hat{s})$ into individual mappings on the symbolic memory. We formalise this function in Section 4.2.

LOAD The LOAD rule is very similar to the corresponding rule in the concrete semantics. The only difference now is that there exists a function $load\_bytes$ that will concatenate n bytes, starting at $k + o$, from the symbolic memory, where n is the size of t. We formalise the function $load\_bytes$ in Section 4.2.

SYMASSERT The SYMASSERT rule will look at the value c on top of the stack $(c, \hat{s}) :: \tilde{st}$. If $c = 0$, it immediately raises *AsrtFail* and the interpreter stops. Otherwise, we have to check if that the current path condition implies that the symbolic expression being asserted is different 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, we check the satisfiability of $\pi \wedge (\hat{s} = 0)$ and, if it is satisfiable, then the assertion fails and the outcome *AsrtFail* produced. Otherwise, the assertion holds and the program may continue, as given by the outcome $\cdot$.

SYMASSUME The SYMASSUME rule analyses the value c on top of the stack. This instruction is used to state that the symbolic expression associated with the value on top of the stack, $\hat{s}$, is assumed to be different from 0. Hence, if the value c is equal to 0, the concrete values of the current execution do not satisfy the programmer's assumptions, which means that the current concolic iteration can be discarded because 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 $(\hat{s} = 0)$. Suppose the value c on top of the stack is different from 0. In that case, the concolic execution may proceed with the current concolic iteration, 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 logical environment, $\hat{x} \in \varepsilon$, then this variable already exists and its mapped value is inserted on top of the stack $(\varepsilon(\hat{x}), \hat{x}) :: \tilde{st}$. Suppose this variable does not exist in the logical environment. In that case, a new entry is added to the logical environment, where $\hat{x}$ is mapped to a random value i of type t, resulting in the new logical environment $\varepsilon' = \varepsilon[\hat{x} \rightarrow i]$, and $(i, \hat{x})$ being put on top of the stack.

4.1.2 Concolic loop

Algorithm 4.1 Concolic Interpreter main loop.

```
1: function CONCOLICEXECUTE(e, $\tilde{\rho}$, $\tilde{st}$, $\tilde{\delta}$, $\tilde{\mu}$)
2:     $\Pi \leftarrow true$
3:     while $\Pi$ is SAT $\wedge$ belowLimit do
4:         $\varepsilon_i \leftarrow$ Model($\pi$)
5:         $e, \tilde{\rho}, \tilde{st}, \varepsilon_i, true, \tilde{\delta}, \tilde{\mu} \Rightarrow_{cs} \tilde{\rho}', \tilde{st}', \varepsilon', \pi', \tilde{\delta}', \tilde{\mu}', \tilde{o}$
6:         if $\tilde{o} \neq$ Error then
7:             $\Pi \leftarrow \Pi \wedge \neg\pi'$
8:         else
9:             return false
10:        end if
11:    end while
12:    return true
13: end function
```

Concolic execution engines execute a given program multiple times in order to explore all possible execution paths. Algorithm 4.1 presents the main analysis loop, which is executed in the concolic interpreter. 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 logical 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.

4.2 Symbolic Memory

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 4.2, 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 4.1. 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 type is used for storing different data types in the same memory location; **(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.

```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;
}
```

Listing 4.1: Symbolic memory manipulation example, taken from [3].

```
(func $swap (param $x i32) (result i32)
  local.get $src
  local.get $x
  i32.store
  local.get $dst
  local.get $src
  i32.load8_u offset=0
  i32.store8  offset=3
  local.get $dst
  local.get $src
  i32.load8_u offset=1
  i32.store8  offset=2
  ;; ...
  return)
```

Listing 4.2: Wasm code from Listing 4.1.

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 will copy 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 consisting of the four bytes of `dst`.

Note that the same segment 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. Hence, this example clearly demonstrates the need for byte-level reasoning in symbolic execution tools for low-level languages.

Byte-Level Operators In order to reason about byte-level memory operations, we make use of two dedicated operators $concat$ and $extract$, such that:

- the expression $concat(\hat{s}_1, \hat{s}_2)$ denotes the bit-vector resulting from the concatenation of the bit vectors denoted by $\hat{s}_1$ and $\hat{s}_2$. For instance, it holds that $concat(\texttt{0x0000}, \texttt{0xBEEF}) = \texttt{0x0000BEEF}$.

- the expression $extract(\hat{s}, h, l)$ denoting the bit-vector corresponding to the extraction of the bytes of the bit-vector denoted by $\hat{s}$ that occur between the limits l and h. This means that the expression $extract(\hat{s}, h, l)$ denotes a bit-vector of size n, where $n = h - l$. For instance, it holds that $extract(\texttt{0x0000BEEF}, 1, 0) = \texttt{0xEF}$.

Given that our underlying Z3 encoding represents all primitive types as bit-vectors, the encoding of these operators into the logic of Z3 is straightforward 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 symbolic expression 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 $store\_bytes(\tilde{\mu}, l, (c, \hat{s}))$, 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 $\tilde{\mu}$ pointed to by the l, resulting in a new symbolic memory $\tilde{\mu}'$.
- The function $load\_bytes(\tilde{\mu}, l, n)$, that sequentially loads n concrete and symbolic bytes from the concolic memory 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

STOREBYTES

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

LOADBYTES

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

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 $concat(\hat{s}_1, \hat{s}_2)$. 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 simplification to symbolic values, every time a symbolic value is loaded from memory. The proposed simplification rules are given below.

Simplification 4.2.1 (Symbolic Extract rule)**.** Given a symbolic expression $\hat{s}$ denoting a value of type t,

the following implication holds:

$$h - l = size(t) \Rightarrow extract(\hat{s}, h, l) = \hat{s}$$

Simplification 4.2.2 (Symbolic Concat rule)**.** For any symbolic expression $\hat{s}$, it holds that:

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

Back to the example Let us now consider the execution of the function `swap` with a symbolic integer `x`. Figure 4.3 illustrates the symbolic content of the C stack: **(a)** immediately before the execution of line 7 and **(b)** immediately before the return statement.

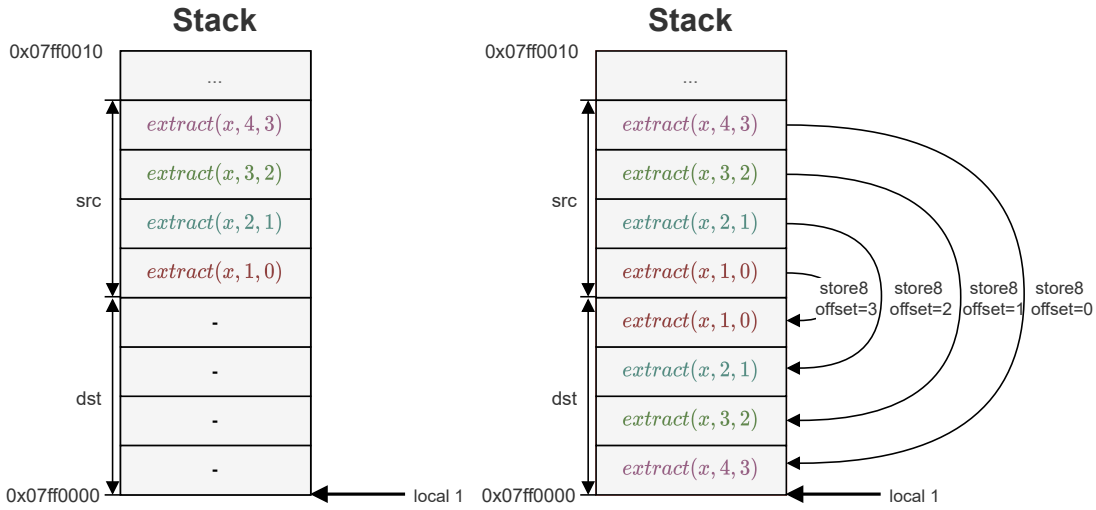


Figure 4.3: Stack memory layout for the program in Listing 4.2. In particular: in (a) we represent the stack before the execution of function `swap`, and in (b) the stack after the execution of the function `swap`.

First, in line 6, we store a 32-bit symbolic integer using the STORE rule, causing four symbolic bytes to be inserted into the memory segment referenced by `src`. Each symbolic byte is represented by the expression $extract(x, i + 1, i)$ with $i = 0, \ldots, 3$. The most significant byte of `src` corresponds to $extract(x, 4, 3)$, the second most one to $extract(x, 3, 2)$ and so on and so forth. Then, the code performs a sequence of four loads followed by stores, essentially copying each byte of `src` to the segment of memory referenced by `dst` in reverse order. For instance, in line 7, the symbolic expression denoting the first byte of `src` is pushed onto the top of the stack; then, in line 8, that expression is copied to the memory cell corresponding to the fourth byte of `dst`. The same reasoning applies to the second, third, and fourth bytes of `src`, which are respectively copied to the segments of memory corresponding to the third, second, and first bytes of `dst`. Note that before copying each byte, the program starts by pushing the addresses of `dst` and `src` to the top of stack as they are required by the store and load operations.

4.3 First Order Solver

The symbolic interpreter of WASP must interact with Z3: **(1)** to generate new concrete inputs at the end of each concolic iteration by checking the satisfiability of the global path condition; and **(2)** to check the validity of user-supplied assertions by checking the satisfiability of their negation. Every time WASP needs to check the satisfiability of a given formula, it first converts the formula into a native Z3 formula and then queries Z3 for the satisfiability of the obtained formula. WASP includes a first order solver whose job is exactly to encode WASP formulas into the logic of Z3.

Our Z3 encoding models Wasm 32-bit and 64-bit integers respectively as Z3 32-bit and 64-bit bit-vectors and 32-bit and 64-bit floats as Z3 single-precision and double-precision floats. Importantly, Z3 floats are internally represented as bit-vectors, streamlining the conversion between both types of values. Wasm binary and unary operators are encoded into equivalent Z3 operators for each type, i.e., binary operators for integers are encoded as Z3 bit-vector operators and float operators as Z3 floating-point operators. When no equivalent Z3 operator exists, the operator is mapped to an uninterpreted function.

Let us now consider how WASP encodes its formulas into the logic of Z3. Table 4.1 displays three representative examples of simple formulas gathered during concolic execution. For each Wasm formula, we specify the Wasm value types of the symbolic inputs, the corresponding type (sort) in the logic of Z3, and the resulting Z3 formula.

Wasm Types	Wasm Formula	Z3 Sort	Z3 Formula
$x, y \mapsto i32$	$y > x$	$x, y \mapsto$ `(BitVec 32)`	`(bvsgt y x)`
$u \mapsto i64$, $v \mapsto i32$	$v = extract(u, 8, 4)$	$u \mapsto$ `(BitVec 64)` $v \mapsto$ `(BitVec 32)`	`(= v ((extract 63 32)u)))`
$z, w \mapsto f32$	$z \leq w$	$z, w \mapsto$ `(Float32)`	`(fp.leq z w)`

Table 4.1: Z3 encoding examples.

Once a formula is encoded into Z3 logic, WASP asks Z3 whether or not it is satisfiable. This can lead to three possible outcomes: **(1)** Z3 concludes that the formula is satisfiable (SAT), and a model is returned; **(2)** Z3 concludes that the formula is unsatisfiable (UNSAT), and no model is returned; **(3)** Z3 cannot determine the satisfiability of the formula (UNKNOWN), and the execution stops.

Every time a formula is found to be satisfiable, WASP requires the model that witnesses its satisfiability. When checking the validity of a user-supplied assertion, the returned model corresponds to the concrete counter-model that triggers the assertion failure. When checking the satisfiability of the global path condition, the returned model corresponds to the logical environment with which to start the next concolic iteration.

Although Z3 does return models of satisfiable formulas, the models generated by Z3 must then be

lifted from the logic of Z3 into the logic of WASP. For instance, 32/64-bit bit-vectors must be converted to Wasm 32/64-bit integers, and single/double precision Z3 floats must be converted to Wasm floats. To this end, we use a lifting function that essentially creates the values of the appropriate type by parsing their string representation generated by Z3.

Let us now consider how WASP lifts a model from the logic of Z3 into its logic. Table 4.2 displays three possible Z3 models for the respective queries presented in Table 4.1. For the first two models, Z3 returns concrete bit-vectors in a hexadecimal representation, which WASP can straightforwardly convert into integers by parsing their string representation. In the last model, Z3 returns a model of two concrete IEEE 754 single-precision floating-points as a sequence of three bit-vectors that denote, respectively, the sign bit (#b1), the exponent (#x98), and the mantissa (#b00000000000000000000000). WASP's decoding concatenates the three bit-vectors and parses the resulting bit-string as a Wasm 32-bit float.

Original Formula	Z3 Model	Wasm Model
$y > x$	$x \mapsto$ #x20000000, $y \mapsto$ #x20000001	$x \mapsto 536870912$, $y \mapsto 536870913$
$v = extract(u, 8, 4)$	$u \mapsto$ #x0000008000000000, $v \mapsto$ #x00000080	$u \mapsto 549755813888$, $v \mapsto 128$
$z \leq w$	$z, w \mapsto$ (fp #b1 #x98 #b00000000000000000000000)	$z, w \mapsto -1.86264514923e^{-09}$

Table 4.2: Z3 decoding examples.

4.4 Restarts

Programmers often need to test their functions not for all inputs but only for those that satisfy a specific set of constraints. To do this, programmers declare symbolic input variables and write assumptions that constrain the values that those variables can denote according to the specification under test. In WASP, this is achieved with the **sym_assume** instruction, which filters out all execution paths for which the symbolic inputs do not satisfy the given constraints.

As explained in Section 4.1.1, 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 Listing 4.3. 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 4.4.

```
int main() {
  int x = symbolic("x");
  int y = symbolic("y");
  int z = symbolic("z");
  int w = symbolic("w");
  int u = symbolic("u");
  sym_assume(x >= 0);        // Asm_1
  sym_assume(y >= 0);        // Asm_2
  sym_assume(z >= 0);        // Asm_3
  sym_assume(w >= 0);        // Asm_4
  sym_assume(u >= 0);        // Asm_5
  sym_assume(x != y);        // ...
  sym_assume(x != z);
  sym_assume(x != w);
  sym_assume(x != u);
  sym_assume(y != z);
  sym_assume(y != w);
  sym_assume(y != u);
  sym_assume(z != w);
  sym_assume(z != u);
  ...
  sym_assume(w != u);        // Asm_n
}
```

Listing 4.3: Chained assumptions that do not affect the path condition set.

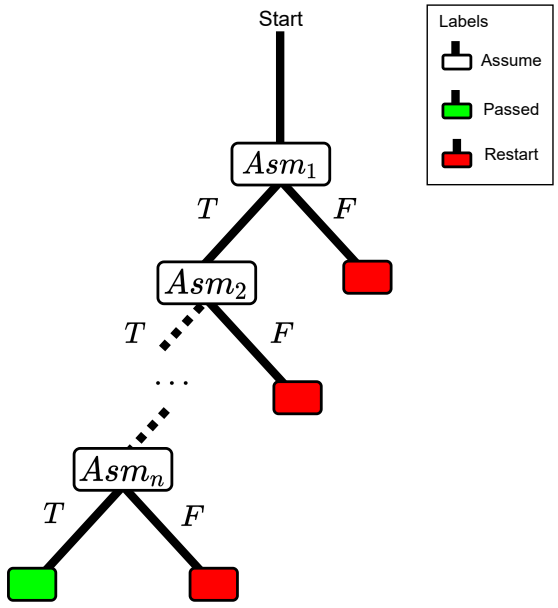


Figure 4.4: Execution tree for the example in Listing 4.3.

To solve this problem, we propose a simple adaptation of the concolic semantics of SymAssume given in Figure 4.2, which avoids the need for restarting the concolic execution 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

SymAssume-Fail

$$\frac{c = 0 \qquad (\pi \wedge \hat{s}) \text{ UNSAT}}{\textbf{assume}, ((c, \hat{s}) :: \tilde{st}), \pi \ \Rightarrow_{\text{cs}} \ \tilde{\rho}, (\neg\hat{s} \wedge \pi), \text{AsmFail}}$$

SymAssume-Pass1

$$\frac{c \neq 0}{\textbf{assume}, ((c, \hat{s}) :: \tilde{st}), \pi \ \Rightarrow_{\text{cs}} \ \tilde{st}, (\hat{s} \wedge \pi), \cdot}$$

SymAssume-Pass2

$$\frac{c = 0 \qquad \varepsilon' = \text{Model}(\pi \wedge \hat{s}) \qquad (\tilde{\rho}', \tilde{st}', \tilde{\delta}', \tilde{\mu}') = \text{UpdtModel}(\varepsilon', (\tilde{\rho}, \tilde{st}, \tilde{\delta}, \tilde{\mu}))}{\textbf{assume}, \tilde{\rho}, ((c, \hat{s}) :: \tilde{st}), \varepsilon, \pi, \tilde{\delta}, \tilde{\mu} \ \Rightarrow_{\text{cs}} \ \tilde{\rho}', \tilde{st}', \varepsilon', (\hat{s} \wedge \pi), \tilde{\delta}', \tilde{\mu}', \cdot}$$

SymAssume-Fail This rule is analogous to its previous version given in Figure 4.2. 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.

```
1  (func $main
2    ;; initialize x=symbolic(), y=symbolic(),
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))
```

Listing 4.4: Wasm module for the program in Listing 2.4.

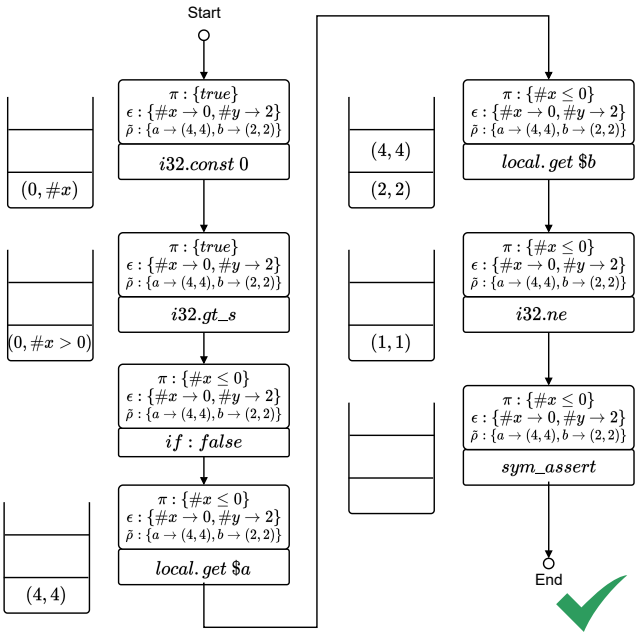


Figure 4.5: Concolic execution of Wasm program from listing 4.4. first execution path.

SYMASSUME-PASS1 This rule is identical to its previous version given in Figure 4.2, so we omit the explanation.

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 logical 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 logical environment, ε'. To this end, we make use of a dedicated function UpdtModel that receives as input a logical environment ε' and a concolic state, generating a new concolic state, obtained by updating the concrete components of the input state according to the supplied logical environment.

4.5 Running Example

To illustrate the concept of concolic execution in WASP, we recall the concolic example in Section 2.2. Recall that for the example in Listing 2.4 an assertion violation was triggered for the inputs $x = 1$ and $y = 4$. Now we will show how WASP can be used to find this assertion violation and generate an appropriate counter model. To this end, we now run through the program in Listing 4.4.

The program shown in Listing 4.4 has two symbolic variables and depending on their values will update variables “a” and “b”. At the end of the program an assertion is made that checks if the two variables differ. For an assertion violation to occur both variables must be updated, this means that the `then` condition of both if-instructions is executed. Figures 4.5 and 4.6 illustrate the step-by-step WASP execution of the given program, where WASP explores two of the possible execution paths: the else path of the first `if` in Figure 4.5, and the path that triggers the assertion violation in Figure 4.6.

Figures 4.5 and 4.6 omit the memory and global store components of the WASP configuration, as they are not used by the program. We graphically represent the remaining components of WASP states using two visual elements: a stack on the left and a box on the right containing the executed instruction, the path condition (π), the logical environment (ϵ), and the local store ($\tilde{\rho}$).

In Figure 4.5, WASP explores the “else” path of the first if of the program and, using a SMT solver, determines that the assertion holds and terminates execution of the current iteration. Next, WASP negates the set of path conditions computed so far and, using Z3, obtains a model that will force execution of the “then” branch of the first if of the program. Figure 4.6 depicts the second iteration of the concolic loop, which now executes the “then” branch of the if statement and updates the variable “b”. Next, since “y” is initialised with an integer bigger than “x”, it executes the “then” branch of the second if statement, updating the variable “a” to $2 \times \#x + \#y$. Lastly, the assert instruction is executed. As part of its execution WASP queries Z3 for the formula

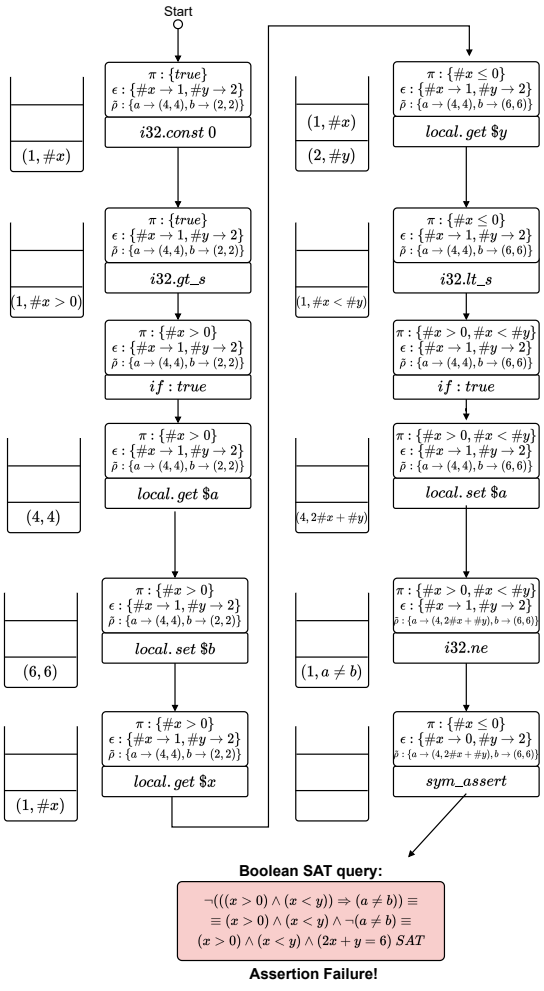


Figure 4.6: Concolic execution of Wasm program from Listing 4.4. Assertion violation path.

$(x > 0) \wedge (x < y) \wedge (2 \times \#x + \#y = 6)$, for which Z3 is able to find the assignment $\#x = 1$ and $\#y = 4$, leading to an assertion violation.

Although we present two possible execution paths, it may be the case that only one path is executed in order to prove that the assertion fails. Notice that, since the values assigned to symbolic variables in the first concolic iteration are random, there is no guarantee which path is going to actually be explored by WASP.

Summary

This chapter formalises the execution engine at the core of WASP, and explains how it can interact with its underlying constraint solver and concolic domains. We further describe our symbolic model for Wasm linear memory, how we encode Wasm formulas into the logic of Z3, and an optimised version of the semantics for the execution the assume instruction. We finish with an illustrative example how WASP works in practice.

5

WASP-C

Contents

This chapter presents WASP-C, a symbolic execution framework to symbolically test C programs using WASP. We begin by giving an overview of the processes required to take a C program and transform it into a Wasm module for WASP to analyse (Section 5.1). Then, we explain how we handle system-level interactions and library calls (Section 5.2). Lastly, we present efficient test-generation techniques for C programs (Section 5.3).

5.1 Overview and Implementation

WASP-C is a symbolic execution framework that concolically analyses programs written in C. WASP-C takes as input C programs annotated with assumptions and assertions and outputs a *test suite*. A 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. However, since WASP does not directly operate over the C source code, WASP-C defines 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*, according to the high-level architecture described in Figure 5.1.

Let us now take a look at how WASP-C concolically executes C programs using WASP as a submodule. First, the *C Pre-processor* parses the given program using a standard C parser called *pycparser* [1], generating an abstract syntax tree (AST) that is then sent to a specialised C visitor. Our specialised C visitor traverses the AST, replacing binary operators such as logical ANDs and ORs with specific function calls. Then the AST is exported back to a C program, which is subsequently compiled into Wasm by the *Compilation Module*. Lastly, the *Wasm Post-processor* processes the obtained Wasm module so as to inject the appropriate WASP symbolic primitives.

In summary the main modules of WASP-C are the following:

1. C Pre-processor: parses the C program received as input and re-writes logical expressions in order to minimise the overlap between the generated test cases as explained in Section 5.3. Additionally, this module handles all IO operations regarding files.

2. Compilation Module: compiles the given C program into Wasm, using the LLVM compilation pipeline described in Section 2.1.3. As a part of the compilation, this module includes a static (partial) implementation of Libc consisting of a set of symbolic summaries described in Section 5.2.

3. Wasm Post-processor: processes the obtained Wasm code to replace calls to mock functions representing the WASP symbolic primitives with their corresponding instructions.

[1] https://github.com/eliben/pycparser

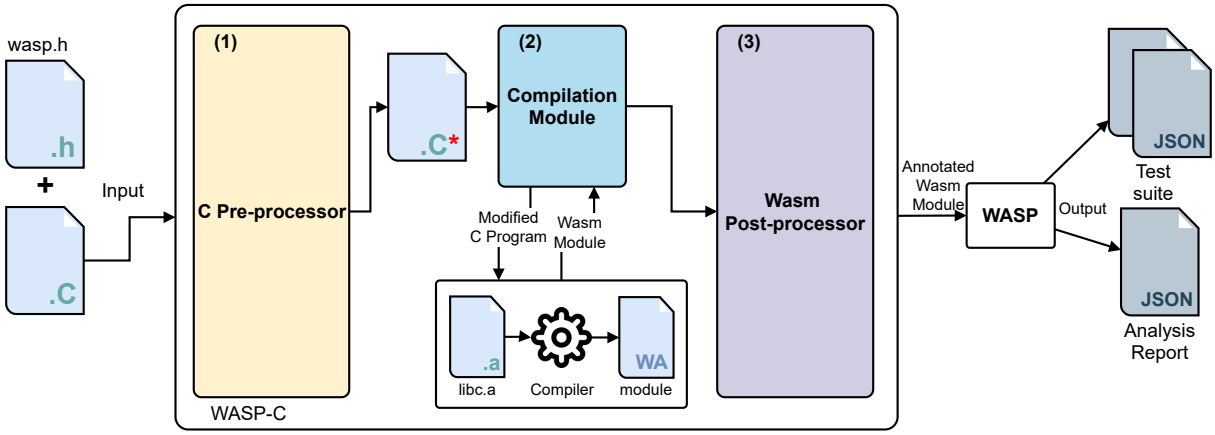


Figure 5.1: WASP-C high-level architecture.

```c
/* test.c */
#include <wasp-c.h>
int main() {
  int a = __WASP_symb_int("a");
  __WASP_assume(-128 >= a);
  __WASP_assume(a <= 127);
  int r = a + 1;
  __WASP_assert(r > a);
}
```

Listing 5.1: Example of client program using the WASP-C's symbolic library presented in Listing 5.2.

```c
/* symbolic values */
// I32
int       __WASP_symb_int(char *);
// I64
long long __WASP_symb_llong(char *);
// F32
float     __WASP_symb_float(char *);
// F64
double    __WASP_symb_double(char *);
/* symbolic variable manipulation */
void __WASP_assume(int);
void __WASP_assert(int);
...
```

Listing 5.2: Modelling symbolic programs using mockup functions defined in the library `wasp.h`.

Mock Symbolic Functions for C In order to allow for symbolic testing at the C level, WASP-C exposes a library of symbolic functions for creating symbolic inputs and reasoning over those inputs. Each of these functions is associated with a symbolic primitive at the Wasm level—for instance, the function `__WASP_symb_int` is associated with the primitive `i32.symbolic`. Hence, at post-processing time, calls to these functions are replaced with their corresponding Wasm symbolic instruction–for instance, calls to `__WASP_symb_int` are replaced with the instruction `i32.symbolic`. However, in order for LLVM to successfully compile the given C program, we have to provide mock implementations for our symbolic functions. These mock implementations are discarded at the Wasm level.

With our library of symbolic functions, we can create symbolic tests, such as the one given in Listing 5.1. This program creates a symbolic variable "a" using the symbolic function `__WASP_symb_int`, declares two assumptions over "a" using the symbolic function `__WASP_assume`, and tests that the variable to be returned satisfies a given constraint using the symbolic function `__WASP_assert`.

Listing 5.2 shows a subset of the symbolic functions that WASP-C supports. For instance:

- The functions `__WASP_symb_int`, `__WASP_symb_float`, and `__WASP_symb_double` are respectively used to create symbolic integers, and 32- and 64-bit floating-point numbers.
- The functions `__WASP_assume` and `__WASP_assert` are used to express assumption and assertions over the symbolic values computed by the program.

Besides the ones shown in the figure, WASP-C offers a variety of other symbolic functions for creating and manipulating first-order formulas, which mostly coincide with the primitives supported by WASP.

5.2 Runtime models

As mentioned in Section 3.2, a symbolic execution engine must accurately model the side effects of interactions with the surrounding software stack. For this reason, we modelled system-level interactions in WASP through symbolic summaries that capture their corresponding side effects. For the C standard library, we used *LibcSummaries*, a library of symbolic summaries developed in the context of a parallel MSc thesis [56]. This library includes symbolic summaries for twenty *Libc* functions, including string manipulation functions (e.g., `strlen`, `strcmp`, and `strcpy`), number parsing functions (e.g., `atoi`), and Input/Output (IO) functions (e.g., `fgets`, `getchar`, `puts`, and `putchar`).

The symbolic summaries in LibcSummaries use a symbolic reflection API consisting of a set of symbolic primitives for explicitly manipulating C symbolic states in a tool-independent way. Hence, in order for us to use LibcSummaries in the context of WASP-C, we had to extend WASP with support for the required symbolic primitives.

In order to better understand how symbolic primitives work, let us consider the symbolic summary of `strlen` included in LibcSummaries and given in Figure 5.3. This summary makes use of four symbolic primitives:

- `summ_is_symbolic` that takes an address of a value with n bytes and returns a boolean indicating if that value is symbolic or not.
- `_solver_NEQ` that takes two addresses of values with n bytes and returns the evaluation of the not equal ($\neq$) operator between the two values.
- `_solver_is_it_possible` that asks the underlying solver if the given formula is satisfiable, returning a boolean with the solver's answer.
- `summ_assume` that extends the current path condition with the provided formula and does not return a value.

```c
int strlen(char* s) {
  int i = 0;
  char zero = '\0';
  while(1) {
    if(summ_is_symbolic(&s[i], 1)) {
      if(!_solver_is_it_possible(
            _solver_NEQ(&s[i], &zero, 1)))
        break;
      summ_assume(_solver_NEQ(&s[i], &zero, 1));
    } else if(s[i] == '\0')
      break;
    i++;
  }
  return i;
}
```

Listing 5.3: Example of the summary for `strlen` in LibcSummaries.

```c
#include<wasp.h>
#include<libc_summaries.h>
int main(void) {
  /* Nondet inputs */
  int len;
  char *str;
  str    = malloc(5);
  str[4] = '\0';
  len    = strlen(str);
  __WASP_assert(len == 4);
  return 0;
}
```

Listing 5.4: Example of `strlen` applied to a non-deterministic string null terminated.

The `strlen` summary receives as input a character array possibly containing symbolic characters and returns a symbolic expression denoting the length of the character array; that is, the number of characters until the null character `'\0'`. To this end, this summary traverses the array, checking at each index whether or not it contains a concrete `'\0'`. If that is the case, the summary returns the current index i. If the character at the current index is symbolic, the summary must reason about its possible concrete values. If the symbolic character must denote the concrete null character (e.g. it is not possible that it is different from the null character), the summary returns the current index. If not, then the summary assumes that the current character is different from the null character and proceeds to the next character. Note that this summary is not sound in that it does not model all the possible concrete paths of the real `strlen` function. However, it is precise; that is, all the paths modelled by the summary correspond to concrete paths of the real `strlen` function.

Let us now take a look at how we can use the summary of `strlen` to reason about programs that call `strlen`. Figure 5.4 shows a simple C program that allocates a string in the heap and uses the `strlen` summary to compute its length. First, the program allocates an array on the heap with space for five characters. Then, the program sets the last byte of the array to `'\0'`. Next, it calls the `strlen` summary on the created array. Note that, in WASP, the malloc function returns a pointer to a segment of memory initialised with arbitrary symbolic bytes (which might denote the character `'\0'`). Nevertheless, the call to the `strlen` summary will return the concrete value four, since that summary counts the number of elements of the given array until a concrete `'\0'` is found. Consequently, this summary is not sound as it does not consider the case in which the allocated array contains intermediate elements equal to `'\0'`. But, it is precise, as it constrains the current symbolic state, assuming that all the characters in intermediate positions are different from `'\0'`.

Symbolic reflection primitives To accurately model the side effects of the C standard library, WASP has to implement the symbolic reflection primitives of LibcSummaries. These primitives are implemented as mock functions at the C level, and once the C code is compiled down to Wasm, those mock functions are replaced with WASP native symbolic primitives. Hence, we had to extend WASP with native implementations of all the symbolic reflection primitives required by LibSummaries. The following table presents the concolic semantics of two representative symbolic reflection primitives: `summ_is_symbolic` and `_solver_NEQ`.

Symbolic Reflection Primitives Semantic Rules

IsSymbolic

$$\frac{(c', \hat{s}') = load\_bytes(\tilde{\mu}, c, n) \qquad c' = \hat{s}'}{\textbf{summ\_is\_symbolic}, \tilde{\rho}, ((n, \_) :: (c, \_) :: \tilde{st}), \varepsilon, \pi \ \Rightarrow_{\mathsf{cs}} \ \tilde{\rho}, ((0, 0) :: \tilde{st}), \varepsilon, \pi, \cdot}$$

IsSymbolic

$$\frac{(c', \hat{s}') = load\_bytes(\tilde{\mu}, c, n) \qquad c' \neq \hat{s}'}{\textbf{summ\_is\_symbolic}, \tilde{\rho}, ((n, \_) :: (c, \_) :: \tilde{st}), \varepsilon, \pi \ \Rightarrow_{\mathsf{cs}} \ \tilde{\rho}, ((1, 1) :: \tilde{st}), \varepsilon, \pi, \cdot}$$

NEQ-False

$$\frac{(c_1', \hat{s}_1') = load\_bytes(\tilde{\mu}, c_1, n) \qquad (c_2', \hat{s}_2') = load\_bytes(\tilde{\mu}, c_2, n) \qquad c_1' = c_2'}{\textbf{\_solver\_NEQ}, \tilde{\rho}, (c_2, \hat{s}_2) :: (c_1, \hat{s}_1) :: (n, \hat{s}_3) :: \tilde{st}, \varepsilon, \pi \ \Rightarrow_{\mathsf{cs}} \ \tilde{\rho}, (0, \hat{s}_1' \neq \hat{s}_2') :: \tilde{st}, \varepsilon', \pi, \cdot}$$

NEQ-True

$$\frac{(c_1', \hat{s}_1') = load\_bytes(\tilde{\mu}, c_1, n) \qquad (c_2', \hat{s}_2') = load\_bytes(\tilde{\mu}, c_2, n) \qquad c_1' \neq c_2'}{\textbf{\_solver\_NEQ}, \tilde{\rho}, (c_2, \hat{s}_2) :: (c_1, \hat{s}_1) :: (n, \hat{s}_3) :: \tilde{st}, \varepsilon, \pi \ \Rightarrow_{\mathsf{cs}} \ \tilde{\rho}, (1, \hat{s}_1' \neq \hat{s}_2') :: \tilde{st}, \varepsilon', \pi, \cdot}$$

IsSymbolic The IsSymbolic rule takes an address c and the number n of bytes from the top of the stack. Then, the concolic interpreter loads a concrete pair $(c', \hat{s}')$ with n bytes from the symbolic memory at the provided address c. Finally, the rule checks if the concrete pair coincides with its symbolic element, in which case the `summ_is_symbolic` primitive returns false; otherwise, it returns true.

NEQ The NEQ rules pops three concolic pairs $(c_2, \hat{s}_2)$, $(c_1, \hat{s}_1)$, and $(n, \hat{s}_3)$ out of the stack and loads the concolic pairs stored in the addresses c_1 and c_2 of the linear memory. Then the rule simply checks whether or not the two concrete loaded values coincide. If they do, the rule pushes the concolic pair $(1, \hat{s}_1' \neq \hat{s}_2')$ onto the top of the stack. If not, the rule pushes the concolic pair $(0, \hat{s}_1' \neq \hat{s}_2')$ onto the top of the stack.

5.3 Efficient Test Suite Generation

Test suite generation is the primary output of WASP-C. WASP-C generates a test case for every execution path explored by WASP using the final logical environment of each concolic iteration. Recall that

```c
void test(int a, int b) {
  if (a && b) {
    return 1;
  } else {
    return 0;
  }
}
```

Listing 5.5: Simple C program with a logical AND inside an if-statement.

```
(func $test (param $a i32) (param $b i32)
    (local $ret i32)
    block
      block
        local.get $a
        i32.eqz
        br_if 1     ;; a == 0?
        local.get $b
        i32.eqz
        br_if 1     ;; b == 0?
        i32.const 1
        local.set $ret
        br 0
      end
      i32.const 0
      local.set $ret
      br 0
    end
    local.get $ret)
```

Listing 5.6: Wasm code from Listing 5.5.

the obtained logical environments model paths of the input program; therefore, since WASP explores all execution paths up to a given pre-established depth, WASP-C can generate test suites with very high code coverage, provided that it is given enough resources. However, the semantics of C short-circuit evaluation makes it possible for WASP to generate multiple logical environments that actually explore the same execution paths. Here, we propose a methodology for preventing the repeated exploration of the same execution paths, minimising the overlap between the generated test cases and enhancing the tool's overall performance.

Problem To understand how the short-circuit evaluation of C logical operators may result in the repeated execution of the same program paths and, consequently, on the generation of overlapping test cases, let us consider the C program given in Listing 5.5 and the result of its compilation to Wasm given in Listing 5.6. The original C program branches on the result of a logical AND operation between two integer variables. The semantics of short-circuit evaluation mandates that the compilation of this program to Wasm first checks if "a" is equal to zero, in which case the "else" branch is immediately executed. Then, it must check if "b" is equal to zero, in which case the "else" branch is also executed. If neither "a" nor "b" is equal to zero, the "then" branch is executed. Accordingly, the Wasm program given in Listing 5.6 has two if instructions with the first one (line 7) checking if "a" is equal to zero and the second one (line 10) checking if "b" is equal to 0.

The concolic executing of the Wasm program resulting from the compilation of the C program will generate three concolic iterations:

1. During the first concolic iteration, the concrete values associated with the symbolic variables of the

program are picked randomly. Let us consider, for the purposes of this example, that $a = 0$ and $b = 0$. These inputs cause the program to break out of the innermost block in line 7, resuming execution in line 15 (equivalent to executing the "else" branch in the C code) and generating the final path condition $a = 0$.

2. Before the second concolic iteration, WASP-C queries the underlying constraint solver for the symbolic inputs that satisfy the global path condition ($\Pi \equiv (a \neq 0)$). Let us assume that the solver returns the model $a = 1$ and $b = 0$. These inputs cause again cause the program to break out of the innermost block in line 10, resuming execution on line 15 (equivalent to executing the "else" branch in the C code) and generating the final path condition $a \neq 0 \wedge b = 0$.

3. Lastly, before the third concolic iteration, WASP-C queries the underlying constraint solver for the symbolic inputs that satisfy the global path condition ($\Pi \equiv ((a \neq 0) \wedge \neg((a \neq 0) \wedge (b = 0)) \equiv (a \neq 0) \wedge (b \neq 0)$). Any such inputs cause the program to execute the innermost block to completion (equivalent to executing the "then" branch in the C code).

Even though the first two concolic iterations explore different paths of the resulting Wasm program, they explore the same path of the original C program. Consequently, WASP-C generates three test cases to obtain 100% code coverage when it only needed two. This problem gets exacerbated when considering sequences of conditional statements, each branching on logical expressions with short-circuit evaluation.

Solution To solve this problem, we perform a pre-processing step right before we compile an annotated C program. This step consists of replacing all AND (`&&`) and OR (`||`) logical operators with function calls to `__logand` and `__logor`, respectively.

To illustrate how the proposed solution helps us avoid the exploration of repeated paths, let us consider the C program given in Listing 5.8, resulting from the application of our pre-processing step to the C program given in Listing 5.5. In the following, we explain the concolic iterations generated by the concolic execution of the Wasm program given in Listing 5.8, which results from the compilation of the pre-processed C program to Wasm.

There are still two possible execution paths. However, contrary to the program given in Listing 5.6, with the proposed solution, there will only be two concolic executions (instead of the three previously observed). WASP executes the given program as follows:

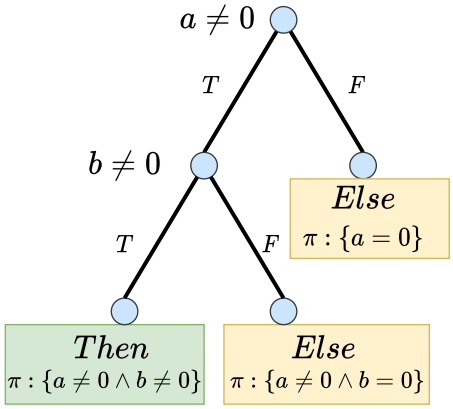
1. During the first concolic iteration, the concrete values associated with the symbolic variables of the program are picked randomly. For this example we consider that $a = 0$ and $b = 0$. These inputs cause WASP to explore the "else" branch of the "if" instruction, generating the final path condition $a = 0 \vee b = 0$.

```c
void test(int a, int b) {
  if (__logand(a, b)) {
    return 1;
  } else {
    return 0;
  }
}
```

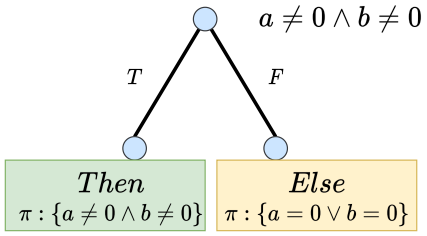
Listing 5.7: Using functions calls instead of short-circuit evaluation.

```
(func $test (param $a i32)
            (param $b i32)
  local.get $a
  local.get $b
  call $__logand
  (if (result i32)
    (then
      i32.const 1)
    (else
      i32.const 0)))
```

Listing 5.8: Wasm code from Listing 5.7.



(a) Original program in Listing 5.5.



(b) Simplified program in Listing 5.7.

$$\pi_{original} \Rightarrow \pi_{simplified}$$

Figure 5.2: Evaluation trees for the original and simplified programs.

2. Before the second concolic iteration, WASP-C queries the underlying constraint solver for the symbolic inputs that satisfy the global path condition ($\Pi \equiv \neg(a = 0 \vee b = 0) \equiv (a \neq 0 \wedge b \neq 0)$). Let us assume that the solver return the model $a = 1$ and $b = 2$. These inputs cause WASP to explore the "then" branch of the "if" instruction, generating the final path condition $a \neq 0 \wedge b \neq 0$.

Lastly, before the third concolic iteration, WASP queries the underlying constraint solver for the symbolic inputs that satisfy the global path condition ($\Pi = ((a \neq 0 \wedge b \neq 0) \wedge \neg(a \neq 0 \wedge b \neq 0))$). Since the condition is unsatisfiable, WASP terminates.

In conclusion, the advantage of modelling the AND logical operator as a function call is that when we reach the `if` statement, we have the full symbolic expression that models the condition. As a result, in the subsequent iterations, WASP can generate logical environments to explore other paths more accurately.

Summary

This chapter introduced WASP-C, a symbolic execution framework we developed to test C programs using WASP. We began with an overview of how to model symbolic programs, followed by the steps required to analyse a C program in WASP-C. Then, we explained how WASP-C models runtime interactions and library calls. Lastly, we presented a strategy to improve test suite generation in WASP-C. The following chapter evaluates the artefact produced until now.

6

Evaluation

Contents

In this chapter, we evaluate the built concolic execution engine to confirm that it is both scalable and able to identify execution errors in actual Wasm modules. We evaluate WASP with respect to three research questions: (RQ1) Can WASP-C be used to detect bugs in C data structure libraries? (RQ2) How does WASP-C support different types of symbolic reasoning? (RQ3) Can WASP-C scale to industry-grade code? Next, we explore each of these research questions and provide our findings in independent sections.

6.1 RQ1: Can WASP-C be used to detect bugs in C data structure libraries?

To investigate whether WASP-C can detect bugs in complex C data structure libraries, we used our tool to symbolically test *Collections-C* [1], a generic data structure library obtained from Github, 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 lines of code (LOC). Next, we present the symbolic test suite we use to evaluate WASP-C on Collections-C (Section 6.1.1) and describe our experimental set-up (Section 6.1.2). Lastly, in Section 6.1.3, we present the results of our experiments.

6.1.1 Benchmark suite

The symbolic test suite we used to evaluate WASP on Collections-C came from the Gillian project [15]. In particular, the authors of Gillian use Gillian-C, their symbolic execution tool for C, to symbolically test Collections-C. To this end, the authors developed a symbolic test suite[2] that they run against the implementation of Collections-C. This symbolic test suite consists of symbolic test program targeting the various data structure algorithms included in Collections-C. Specifically, this symbolic test suite contains 161 symbolic tests.

We test two different versions of Collections-C, a version with bugs previously found by the authors of Gillian-C[3], henceforth buggy version, and the version resulting from the correction of those two bugs[4], 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.

Running program analysis on this symbolic test suite allows us to examine and compare the *precision* and other key performance metrics of our tool. These metrics include *total execution time*, which is the

[1] https://github.com/srdja/Collections-C
[2] https://github.com/GillianPlatform/collections-c-for-gillian
[3] https://github.com/srdja/Collections-C/pull/119 and https://github.com/srdja/Collections-C/pull/123
[4] https://github.com/srdja/Collections-C

time it takes to analyse a test, *concolic loop execution time* representing the time spent running the concolic interpreter and *solver execution time* which gives us the time spent in the constraint solver.

6.1.2 Experimental setup

Benchmark preparation The symbolic test suite already comes annotated with symbolic inputs and assume/assert declarations. However, because the symbolic test suite was designed to be analysed in Gillian-C, the annotations use a syntax not supported in WASP-C. For this reason, these annotations had to be adapted into the functions that we use to symbolically test C programs. For instance, the declaration of symbolic integers must be re-written as follows:

```
int a = __builtin_annot_intval("symb_int", a);
                        ⇓
int a = __WASP_symb_int("a");
```

This transformation is trivial and is achieved by explicit manipulation of the program text using regular expressions. The remaining Gillian-C symbolic operators are analogously mapped to WASP-C equivalent ones.

Experimental procedure We performed two experiments:

- Experiment one: We use Gillian-C and WASP-C to execute the symbolic test suite on the corrected version of Collections-C.

- Experimento two: We use Gillian-C and WASP-C to execute the two error triggering symbolic tests on the buggy version of Collections-C.

The experiments were conducted 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 and the flag `-m 10_000_000` which limits concolic execution to 10M Wasm instructions. Note that we can only disable WASP's type checker because compilation assures that the instructions are well-typed. Additionally, as a safeguard measure, we set a timeout of twenty seconds, seeing that WASP may get stuck in Z3 during constraint solving.

Category	n_i	GILLIAN-C T_{Gil} (s)	WASP-C 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

Table 6.1: Results for Gillian-C and WASP-C applied to Gillian-C corrected version of Collections-C.

6.1.3 Results

Table 6.1 presents the results of experiment one, 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 analysis time for WASP (T_{WASP}), 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). Note that the concolic interpreter queries the solver at the end of each iteration, meaning T_{solver} is already accounted in T_{loop}. The same is true for T_{loop} regarding T_{WASP}. To help illustrate how T_{loop}, T_{solver}, and T_{WASP} relate to one another. We express these relations arithmetically:

$$T_{WASP} = T_{loop} + T_{parse} \tag{6.1}$$

$$T_{loop} = T_{solver} + T_{interpretation} \tag{6.2}$$

In the formulas above, T_{parse} represents the total time needed to parse the benchmarks and $T_{interpretation}$ the total time spent interpreting Wasm instructions.

From Table 6.1 we observe that, overall, WASP is $1.14\times$ faster than Gillian-C at analysing the complete benchmark suite. In 7 out the 10 categories, WASP completes the program analysis faster than Gillian-C (i.e., $T_{WASP} < T_{Gil}$). For Stack and RingBuffer categories, WASP spends a negligible amount of time in the solver ($T_{solver} \approx 0.00$), being able to complete the analysis in a single concolic iteration. Two reasons explain this fact. First, the data structures are simple, i.e., the order of the stored values does not depend on the symbolic variables it contains. For example, a Stack only pushes and pops symbolic values from an array, and a RingBuffer reads and writes symbolic values to an array while

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

Table 6.2: Bug-finding statistics for Collections-C bugs by WASP and Gillian-C.

updating concrete index values. As a result, the concolic interpreter does not add constraints to the path condition set. Second, the test assertions are trivial. For example, when we want to check the consistency of values in a stack, we pop a symbolic value off the stack and compare it with the symbolic input originally inserted onto the stack, generating formulas like $x = x$ or $y \neq y$, which WASP can easily solve without the help of a solver.

Table 6.2 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 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 number of concolic iterations(n_{paths}), the time spent in the concolic loop (t_{loop}), the time spent in the solver (t_{solver}), analysis time for WASP (t_{WASP}), the analysis time for Gillian-C (t_{Gil}), and the speedup between T_{Gil} and T_{WASP} (S). The results demonstrate that that WASP outperforms Gillian-C, being able to complete analysis 3.28$\times$ faster. We believe that this performance gain is naturally 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. Additionally, our second experiment demonstrates that WASP can find the two known bugs previously found by Gillian-C, a state-of-the-art symbolic execution tool for C.

New Bug Found: Besides the two known bugs found by Gillian-C, WASP-C additionally found a new heap overflow bug in the Pqueue category. We confirmed the bug with a concrete test using *AddressSanitizer* [57]. In order to explain the discovered bug, let us consider the code of `pqueue_push` given in Listing 6.1. This function inserts the element `elem` into the priority queue `pq`. A Collections-C priority queue corresponds to the standard max-heap data structure [58]. Essentially, the `pqueue_push` function traverses the array in order to find the index at which the given element is to be inserted. At the end of each iteration, the variable i is set to the index of the current parent and the variable `parent` to the value of the parent of the parent. The index of the parent of the node stored at index i is given by the expression $(i-1)/2$. Hence, if one starts the body of the loop with i equal to 1, one will finish the loop with $i = 0$ ($0 = (1-1)/2$). The problem occurs when computing the new value of the variable `parent`, which would be in the index $(0-1)/2$. However, since i is declared to be of type `size_t` (unsigned int), the evaluation of $(0-1)/2$ will trigger an integer overflow, and subsequently lead to a heap overflow in line 12. Even though this bug is harmless, it results in unspecified behaviour, potentially causing porta-

```
1  #define CC_PARENT(x) (x - 1) / 2
2  enum cc_stat pqueue_push(PQueue *pq, void *elem)
3  {
4      size_t i = pq->size;
5      ...
6      while ((i != 0) && pq->cmp(child, parent) > 0) {
7          void *tmp = pq->buffer[i];
8          pq->buffer[i] = pq->buffer[CC_PARENT(i)];
9          pq->buffer[CC_PARENT(i)] = tmp;
10
11         i      = CC_PARENT(i); child  = pq->buffer[i];
12         parent = pq->buffer[CC_PARENT(i)];
13     }
14     return CC_OK;
15 }
```

Listing 6.1: Off-bounds read in the heap by *pqueue_push*.

bility problems. A possible fix would be to add an explicit check for 0 to the macro on line 1. Replacing the current statement for: `#define CC_PARENT(i) ((i > 0) ? (i-1)/2 : 0)`[5].

6.2 RQ2: How does WASP-C support different types of symbolic reasoning?

To investigate our second research question, i.e., how does WASP-C support different types of symbolic reasoning, we test WASP-C against the *Test-Comp benchmark suite* (Test-Comp) [16] and compare its results against those obtained from the testing tools submitted to the 2021 Test-Comp [2]. We first give an overview of the Test-Comp benchmark (Section 6.2.1); then, in Section 6.2.2 we present the experimental set-up required for running WASP-C on the benchmarks of Test-Comp; and finally, in Section 6.2.3 we present the obtained results. The results obtained for WASP-C are comparable to those obtained for well-established symbolic execution testing tools for C, such as KLEE [9] and VeriFuzz [17], demonstrating the maturity of our tool.

6.2.1 Benchmark suite

Test-Comp consists of a collection of programs generated from the C benchmarks used in the "3rd edition of the Competition on Software Testing" (Test-Comp) [2]. Test-Comp utilises benchmarks from *sv-benchmarks* a publicly available benchmark suite of software-verification tasks[6], covering correct and buggy examples. Hence, one can regard Test-Comp as a well-accepted data set for evaluating testing tools for C programs. The benchmarks included in Test-Comp are organised into various categories, with

[5] Bug fix for heap-overflow bug: https://github.com/srdja/Collections-C/pull/148
[6] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

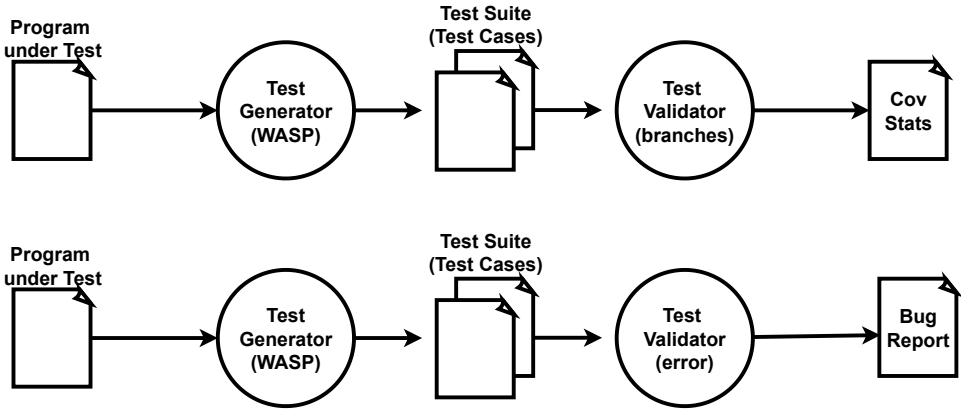


Figure 6.1: Flow of the Test-Comp execution for one tester (taken from [2]).

each category targeting a specific type of symbolic reasoning; for instance, the category *Arrays* aims at reasoning about the treatment of arrays, the category *BitVectors* aims at reasoning about bit-operations, and the category *Loops* aims at testing the analysis of 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 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.

6.2.2 Experimental setup

In order to test WASP-C against the Test-Comp benchmarks, we developed two testing pipelines, one for Cover-Branches tasks and another for Cover-Error tasks. We illustrate these in Figure 6.1. Both testing pipelines receive as input the program to be analysed and use WASP-C to concolically execute it. The difference lies in how the generated output is processed. For Cover-Branches, the testing pipeline feeds all generated test cases to the Test-Comp test validator in branch coverage mode, which outputs the coverage statistics obtained by the test suite. For Cover-Error, the testing pipeline feeds the single bug-triggering test case to the Test-Comp test validator in bug-finding mode, which will check for call coverage of the bug-triggering function to be different from zero.

We separately evaluate WASP-C on the Cover-Branches and Cover-Error tasks. For each task, we assign WASP-C a global score. 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. Table 6.3 summarises the scoring scheme for each type of task. In Cover-Branches, each generated

Points	Description
Cover-Branches	
`+cov` points	for a generated *test suite* that yields coverage `cov` and *time* is less than the time limit
0 points	otherwise
Cover-Error	
+1 point	for a generated *test suite* that contains a test witnessing the specification violation and *time* is less than the time limit
0 points	otherwise

Table 6.3: Test-Comp scoring scheme, taken from [2].

test suite is given a coverage score $cov \in [0, 1]$, while in Cover-Error, a test case is given a score of 1 if it triggers the bug and scores 0 otherwise.

For both tasks, results are presented for each testing category (e.g. Arrays, BitVectors, ControlFLow, and so on.). For each category, we assign a unique integer between 1 and 12. We obtain the global score for all analysed categories by applying a weighted average on the individual scores of each category according to the formula:

$$S = \left(\sum_{i=1}^{k} \frac{sc_i}{n_i}\right) \times \frac{N}{k} \tag{6.3}$$

Where: N is the total number of analysed programs, k is the number of categories, n_i is the number of programs in the category i, and sc_i is the score obtained for category i.

The experimental test bed used to run the two experiments on the Test-Comp benchmark suite was the same test bed used for the evaluation of RQ1 (6.1.2). For the test validator, we adopted TestCov [59] v3.4-dev. For each instantiation of WASP, we use the flag `-u` which disables the WASP type checker. Furthermore, we ran WASP with a timeout of 15 minutes and a 15GiB memory limit to closely replicate the resource constraints imposed during the Test-Comp competition.

6.2.3 Results

Table 6.4 and Table 6.5 presents WASP-C's evaluation results per category for the Cover-Branches and Cover-Error tasks, respectively. In both tables, we compare the results obtained for WASP-C with the tools submitted to Test-Comp 2021: FuSeBMC, CMA-ES Fuzz, CoVeriTest, HybridTiger, KLEE, Legion, LibKluzzer, PRTest, Symbiotic, TracerX, and VeriFuzz. In particular, we show the minimum and maximum recorded scores, the scores corresponding to the first, second and third quartiles of the evaluated tools, and the score and rank obtained by WASP-C. For Cover-Branches and Cover-Error, WASP-C ranked sixth and third, demonstrating that WASP-C's symbolic reasoning is on par with state-of-the-art symbolic execution and testing tools for C. Note that in Table 6.5 there are no results for the categories C11 and C12 because these categories have no Cover-Error tasks. Additionally, we can see

Category	Min	Q1	Q2	Q3	Max	WASP-C	Rank
C1.Arrays	96.0	167.0	225.0	256.5	296/400	245/380	4th
C2.BitVectos	13.0	27.5	37.0	37.5	40/62	35/57	7th
C3.ControFlow	3.0	6.5	15.0	16.0	18/67	33/54	1st
C4.ECA	0.0	2.5	6.0	8.5	12/29	4/27	8th
C5.Floats	16.0	49.0	64.0	94.0	103/226	78/202	7th
C6.Heap	19.0	71.5	81.0	86.0	90/143	80/136	7th
C7.Loops	152.0	272.5	383.0	407.0	424/581	403/572	4th
C8.Recursive	9.0	19.5	31.0	35.5	38/53	27/51	8th
C9.Sequentialized	0.0	17.0	39.0	58.0	71/82	25/39	9th
C10.XCSP	0.0	74.0	80.0	84.5	97/119	56/100	10th
C11.Combinations	0.0	11.5	31.0	117.0	180/210	28/210	7th
C12.MainHeap	51.0	152.0	185.0	196.0	204/231	175/226	8th
Score	411.0	717.5	1087.0	1165.0	1389	1090	6th

Table 6.4: Results for the meta category Coverage-Branches.

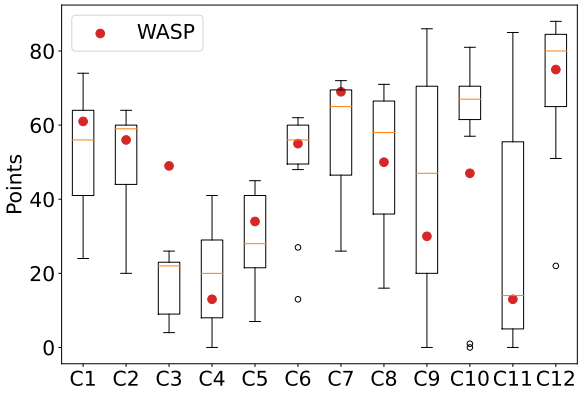
Category	Min	Q1	Q2	Q3	Max	WASP-C	Rank
C1.Arrays	0	63.0	73.0	90.5	96/100	89/100	4th
C2.BitVectors	0	5.5	8.0	9.0	10/10	7/10	2th
C3.ControlFlow	0	3.5	8.0	9.0	11/32	29/32	1st
C4.ECA	0	0.5	2.0	12.5	16/18	7/18	6th
C5.Floats	0	0.0	6.0	26.0	32/33	21/32	5th
C6.Heap	0	23.0	44.0	46.5	47/57	41/55	7th
C7.Loops	0	44.0	82.0	116.5	138/158	127/156	4th
C8.Recursive	0	0.5	13.0	16.5	19/20	9/20	7th
C9.Sequentialized	0	28.5	79.0	89.5	101/107	75/107	9th
C10.XCSP	0	1.5	31.0	43.5	53/59	54/59	1st
C11.Combinations	–	–	–	–	–	–	–
C12.MainHeap	–	–	–	–	–	–	–
Score	0	152.0	266.0	349.0	405	360	3rd

Table 6.5: Scores for each sub-category in the Cover-Error category.

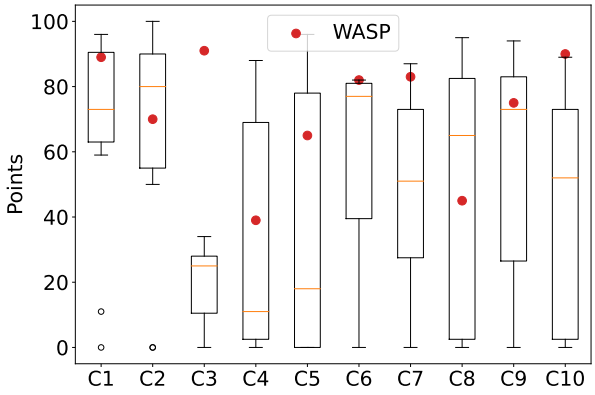
that WASP-C ranked fourth when comparing our tool against the 11 test-generation tools evaluated in Test-Comp 2021.

To help visualise the results from Table 6.4 and Table 6.5 we create Figures 6.2(a) and 6.2(b) that plot the results for Cover-Branches and Cover-Error, respectively, by normalising the scores of all tools in each category. We highlight WASP-C's score with a red dot.

In the Cover-Branches tasks (see Figure 6.2(a)), WASP-C outperforms the other tools for ControlFlow (C3) and performs above average for C1, C5 and C7. In contrast, for XCSP (C10), WASP-C performs worse than other tools due to a combination of three factors: **(1)** the XCSP programs are exclusively made of sequences of assumptions; **(2)** WASP-C only generates satisfying assignments for assumes; and **(3)** the `assume` function in Test-Comp is implemented as an if-statement. Meaning that because (2) and (3) WASP only generates test cases that explore the "then" branch of the `assume` function,



(a) Box plot for Table 6.4



(b) Box plot for Table 6.5

Figure 6.2: Box plots

		Cover-Error			Cover-Branches			Overall		
Rank	Tester	Score	CPU	Rank	Score	CPU	Rank	Score	CPU	Rank
1	VeriFuzz	385	2.6	2	1,389	630	1	1,865	640	1
2	FuSeBMC	405	22	1	1,161	390	4	1,776	410	2
3	LibKluzzer	359	90	4	1,292	520	2	1,738	610	3
4	**WASP-C**	**360**	26	3	**1,090**	310	6	**1,585**	330	4
5	Symbiotic	314	5	6	1,169	440	3	1,543	450	5
6	KLEE	339	3	5	784	210	9	1,370	220	6

Table 6.6: CPU execution time in hours for the top-6 overall ranked testers.

consequently, because of (1) WASP-C can only ever achieve 50% coverage of the entire XCSP category.

In the Cover-Error tasks (see Figure 6.2(b)), WASP-C establishes a new maximum score for ControlFlow (C3) and XCSP (C10) and performs above average for C1, C4 and C7. Contrary to Cover-Branches, in Cover-Error, the fact that WASP can only generate satisfying assignments for `assume` statements helps WASP-C achieve first place in XCSP. Consider the fact that XCSP programs are sequences of assumptions and that to find the assertion violation, one must find the path that satisfies all assumptions. Then, because WASP only generates satisfying assignments for the `assume` statement, WASP quickly progresses through the sequence of assumptions and finds the assertion violation.

Comparing the CPU time over the top-6 ranked tools, Table 6.6 shows that, for Cover-Error, WASP-C was the fifth fastest with a total time of 26 hours. For Cover-Branches, WASP-C was the second-fastest tool with a total of 310 hours, being only beaten by KLEE, which completed analyses in 210 hours. Overall, WASP-C is the second-fastest tool in the top-6, finishing analyses in about 326 hours and placing fourth (note that KLEE placed sixth). Furthermore, it is essential to mention that the tools we compare WASP-C against were executed on a far superior test bed. Test-Comp's test bed is a

compute 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. Compared to our server with a Intel Xeon E5–2620 CPU, a frequency of 2.5GHz and 32GB of RAM, and where we executed twelve test-generation runs concurrently.

6.3 RQ3: Can WASP-C scale to industry-grade code?

To investigate our third research question, i.e., how does WASP-C scale to industry-grade code? We use WASP-C to obtain a comprehensive test suite for part of the C implementation of the AWS Amazon Encryption SDK[7]. We begin by characterising the subset of the library tested by WASP-C (Section 6.3.1), then we explain the experimental setup (Section 6.3.2) and finally present the results of our experiments (Section 6.3.3).

6.3.1 Benchmark suite

The Amazon Web Services (AWS) Encryption Software Development Kit (SDK) for C is a library for the encryption and decryption of data using standards and best practices. For instance, it uses OpenSSL[8] a general-purpose cryptography toolkit used for secure communications.

The AWS Encryption SDK for C is an extensive client-side library that implements complex data structures in the C language. In particular, this library is difficult to analyse as it implements various cryptographic functions that current SMT solvers cannot tackle. This library comes with a benchmark suite of static bounded verification proofs designed to be checked with the *CBMC bounded model checker* [60]. These proofs can be easily turned into symbolic tests, which WASP-C can leverage to generate a concrete test suite for the library. We consider 94 verification proofs totalling 6k LOC. The library itself contains multiple C files totalling just under 40k LOC. The Amazon DynamoDB Encryption Client[9] uses this library to encrypt its client's sensitive table data.

6.3.2 Experimental setup

Each verification proof corresponds to a file containing a C program annotated with non-deterministic data structures and assumptions. For instance, Listing 6.2 presents a fragment of the bounded verification proof for the `cmm_base_init` operation over the `cmm` structure. Before applying WASP-C, we have to convert the verification proof into a symbolic test replacing CBMC bounded verification primitives

[7] https://github.com/aws/aws-encryption-sdk-c
[8] https://www.openssl.org/
[9] https://docs.aws.amazon.com/crypto/latest/userguide/awscryp-service-ddb-client.html

```c
#include <aws/cryptosdk/materials.h>

void aws_cryptosdk_cmm_base_init_harness() {
    /* Nondet input */
    struct aws_cryptosdk_cmm cmm;
    struct aws_cryptosdk_cmm_vt *vtable =
        malloc(sizeof(struct aws_crypto_cmm_vt));

    /* Assumptions */
    __CPROVER_assume(aws_cryptosdk_cmm_vtable_is_valid(vtable));

    /* Operation under verification */
    aws_cryptosdk_cmm_base_init(&cmm, vtable);

    /* Post-conditions */
    assert(aws_cryptosdk_cmm_base_is_valid(&cmm));
}
```

Listing 6.2: Bounded verification proof for the `cmm_base_init` operation over the `cmm` structure.

```c
void ensure_nondet_allocate_cmm_vtable_members(
        struct aws_cryptosdk_cmm_vt *vtable, size_t max_len) {
    if (vtable) {
        vtable->vt_size = __WASP_symb_int("vt_size");
        vtable->name = malloc(sizeof(char) * max_len);
        for (int i = 0; i < max_len; ++i)
            vtable->name[i] = __WASP_symb_char();
    }
}
```

Listing 6.3: Initialisation of the `cmm_vt` structure using WASP-C's primitives

with WASP-C's primitives and explicitly associating each function parameter with a symbolic variable of the appropriate type. For instance, the CBMC's annotation for assumptions `__CPROVER_assume(expr)` is replaced with `__WASP_assume(expr)`. This transformation enables the compilation of the verification proofs.

While parameters of primitive types can be straightforwardly converted to WASP's symbolic inputs, non-primitive parameter types require a more complex approach. In particular, we allocate symbolic structs and arrays into the heap and initialise their components with symbolic values of the appropriate type. For instance, in Listing 6.3 we present the initialisation function that we use for the symbolic `struct cmm_vt` in the example of Listing 6.2. In particular, we explicitly declare two symbolic fields from the struct. First, the size of the struct (`vt_size`) is declared as symbolic using the `__WASP_symb_int` primitive. Then, for the second field, which is a string, we allocate space for it in the heap and initialise it with symbolic characters using the `__WASP_symb_char` primitive.

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 the execution tree of a program; this is important because we are interested in arguing about WASP-C's ability to scale to real-world code applications. Consequently, we expect that for the analysis that do not timeout at 15 minutes, the generated test suite produces high-coverage tests. Furthermore, we expect that the analysis also scales gradually with the gradual increase of complexity in the benchmark.

The experimental test bed used to run the AWS Encryption SDK benchmark suite was the same test bed used for the evaluation of RQ1 and RQ2 (6.1.2 and 6.2.2). For each instantiation of WASP, we use the flag `-u`, which disables WASP's type checker.

6.3.3 Results

Table 6.7 presents the results of our experiments. For each category in the benchmark suite we present: the number of tests in that category (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}), and the total analysis time for that category (T_{global}). We organised the tests into categories by the data structure they are modifying/testing or specific operations like encryption/decryption. Tests for generic data structures like lists or hash tables or generic operations like getters/setters go into the Misc-ops category as they are not specific to the AWS Encryption SDK. In total, WASP-C analyses the benchmark suite in three hours. Not surprisingly, in the table, we observe that for data structures *Md* and *Materials* that contain concrete values are very quickly analysed, while data structures that mainly contain symbolic values like *Edk*, *Enc_ctx*, *Cmm*, *Sig*, *Keyring*, *Private*, and *Hdr* take much more time to analyse. Also, note that analyses finish quickly in the Encrypt/Decrypt categories that test encryption/decryption operations. We believe this is due to the small inputs that the encryption/decryption operations receive. For instance, these operations receive strings with one or two characters at most.

The results demonstrate that WASP-C can be applied to real-world code. In particular, we see that the overall running time of WASP grows linearly with the number of explored paths. Specifically, with the measurements from Table 6.7, we observe that: (i) T_{loop} grows linearly with n-paths, and (ii) T_{solver} grows linearly with n-paths. Concretely, for categories Encrypt and Edk, the number of paths explored increases by roughly one order of magnitude (from 76 to 410); as a result, the T_{loop} and T_{solver} also roughly increase one order of magnitude (from 11.62s to 507.15s and 4.95s to 10.61s, respectively). Then, the same phenomenon happens between Edk and Hdr, when the number of paths explored increases one order of magnitude over Edks's.

We believe that observation (i) is accurate because non-trivial paths through the code are of comparable depths since all loops are bound to a concrete number of iterations. To justify observation (ii), we must first recall from Algorithm 4.1 that we perform conjunction of the negation of the generated path condition with the global path condition set. Then, assuming that observation (i) holds, paths of

Category	n_i	n-paths	T_{loop} (s)	T_{solver} (s)	T_{global} (s)
Md	3	5	0.17	0.06	1.93
Materials	4	8	2.08	0.13	4.02
Encrypt	4	76	9.78	3.94	11.69
Decrypt	4	76	11.62	4.95	13.63
Edk	7	410	507.15	10.61	510.96
Enc_ctx	7	559	903.83	99.03	907.85
Cmm	7	832	903.03	56.08	907.16
Sig	9	1,143	250.42	49.61	255.05
Keyring	17	1,274	1,176.94	144.03	1,186.03
Private	5	1,459	2,689.9	533.4	2,692.87
Hdr	16	2,940	5,405.03	308.74	5,415.04
Misc-ops	11	3,211	1,938.59	132.57	1,944.66
Total	94	11,993	13,798.64	1,343.15	13,850.89

Table 6.7: Benchmark results applying WASP-C to the AWS Encryption SDK for C.

comparable depth will extend the global path condition set with formulas of approximately constant size, which results in linear growth of the global path condition set. Although it is not trivial that linear growth of the global path condition leads to linear growth in constraint solving time, we observe this.

Summary

In this chapter, we evaluated WASP-C. We first examined the effectiveness of WASP-C for finding vulnerabilities in data structure libraries. We used the benchmark suite designed to benchmark Gillian-C. After executing both the set of bug-free and the set of bug benchmarks, we found WASP-C to be highly effective in finding bugs. WASP-C not only was able to identify the same bugs as Gillian-C, but it additionally found another vulnerability not reported by Gillian-C, and that can result in unspecified behaviour. Furthermore, WASP-C was slightly more efficient than Gillian-C during analysis.

Then, we examined the different types of symbolic reasoning supported by WASP-C. We used the Test-Comp benchmark suite, which includes symbolic reasoning types, such as the treatment of arrays, reasoning about bit-operations, floating-point arithmetic, analysis of loops, and recursion. After executing the benchmark suite for the specification *coverage-branches* and *coverage-error-call*, we found that WASP-C ranked sixth and third, respectively. In total, WASP-C ranked fourth in terms of scoring and finished analyses faster than the top-3 ranked tools.

Lastly, we examined WASP-C's capability of scaling to industry-grade code. We used AWS Encryption SDK for C, which is a client-side encryption library. After executing the benchmark suite, we found with some level of confidence that the analysis scales linearly. In total, WASP-C took approximately three hours to complete the analysis. However, we expected this because the benchmark suite has

many non-deterministic values in the data structures. Regardless, our results confirm that WASP-C scales relatively well to industry-grade code.

7

Conclusion

Contents

7.1 Conclusions

WebAssembly (Wasm) is a new binary instruction format imposing its presence in modern Web applications by providing a means to run complex code in the browser efficiently. To the best of our knowledge, there are only two tools for symbolically executing Wasm code: WANA [12] and Manticore [13]. However, these tools were only evaluated on smart contracts, which are fundamentally different from general-purpose software applications, which are often much larger and impose different scale requirements.

We propose WASP, a novel concolic execution engine for testing Wasm modules, and WASP-C, a symbolic execution framework to symbolically test C programs using WASP. A significant advantage of using the concolic discipline [10, 14] when compared to static symbolic execution is that the latter requires less frequent interactions with the underlying solver; essentially, one call to the solver per explored execution path.

According to the evaluation we conclude that:

- WASP-C can detect bugs in complex data structure libraries: WASP tested a widely-used generic data structure library for C and detected three bugs in the benchmarks, including a previously unknown bug.
- WASP-C supports different types of symbolic reasoning: WASP was tested against the Test-Comp [16] benchmark suite and obtained results comparable to well-established symbolic execution and testing tools for C, such as KLEE [9] and VeriFuzz [17]. Concretely, WASP was the third-best tool in the cover-error category, the sixth-best tool in the cover-branches category, and the overall fourth-best tool.
- WASP-C can scale well to industry-grade code: WASP was able to generate a high-coverage test suite for the Amazon Encryption SDK for C[1].

7.2 Future Work

In the future, we plan to extend WASP with support for static symbolic execution and perform an empirical study comparing the fully static approach against the concolic one. We further plan to integrate both strategies in a future version of WASP, which would leverage machine learning algorithms to select the strategy to apply depending on the specific features of the program to be analysed.

Furthermore, we would like to continue evaluating the Amazon Encryption SDK for C. In particular, we plan to analyse the modules `aws_cryptosdk_session` and `aws_cryptosdk_framestate`, which were

[1] https://github.com/aws/aws-encryption-sdk-c

not considered in the evaluation of WASP due to the time constraints of this project.

Finally, we would like to leverage WASP to build symbolic execution engines for other high-level programming languages, such as Java and Rust, which can already be compiled to Wasm using the *JWebAssembly* [61] and *rustc* [62] compilers.

Bibliography

[1] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017*. ACM Press, 2017. [Online]. Available: https://doi.org/10.1145%2F3062341.3062363

[2] D. Beyer, "Status report on software testing: Test-comp 2021," in *Fundamental Approaches to Software Engineering*, E. Guerra and M. Stoelinga, Eds. Cham: Springer International Publishing, 2021, pp. 341–357.

[3] X. Leroy, R. Blazy, and G. Stewart, "The compcert memory model, version 2," 2012.

[4] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*, ser. IoTDI '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 225–236. [Online]. Available: https://doi.org/10.1145/3302505.3310084

[5] M. P. Daniel Lehmann, Johannes Kinder, "Everything old is new again: Binary security of webassembly," in *Proceedings of the 29th USENIX Security Symposium*, 2020.

[6] B. McFadden, T. Lukasiewicz, J. Dileo, and J. Engler, "Security Chasms of WASM," NCC Group, Tech. Rep., 08 2018.

[7] J. Bergbom, "Memory safety: old vulnerabilities become new with WebAssembly," Forcepoint, Tech. Rep., 12 2018.

[8] J. Rüth, T. Zimmermann, K. Wolsing, and O. Hohlfeld, "Digging into browser-based crypto mining," in *Proceedings of the Internet Measurement Conference 2018*, ser. IMC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 70–76. [Online]. Available: https://doi.org/10.1145/3278532.3278539

[9] C. Cadar, D. Dunbar, and D. Engler, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’08. USA: USENIX Association, 2008.

[10] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed automated random testing,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[11] K. Havelund and T. Pressburger, “Model checking Java programs using Java PathFinder,” *International Journal on Software Tools for Technology Transfer*, 1999.

[12] D. Wang, B. Jiang, and W. K. Chan, “Wana: Symbolic execution of wasm bytecode for cross-platform smart contract vulnerability detection,” 2020.

[13] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, “Manticore: A user-friendly symbolic execution framework for binaries and smart contracts,” 2019.

[14] K. Sen, D. Marinov, and G. Agha, “CUTE: A concolic unit testing engine for c,” in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

[15] J. Fragoso Santos, P. Maksimović, S.-E. Ayoun, and P. Gardner, “Gillian, part i: A multi-language platform for symbolic execution,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3385412.3386014

[16] D. Beyer, “International competition on software testing (test-comp),” in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer, M. Huisman, F. Kordon, and B. Steffen, Eds. Cham: Springer International Publishing, 2019, pp. 167–175.

[17] A. Basak Chowdhury, R. K. Medicherla, and V. R, “Verifuzz: Program aware fuzzing,” in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Beyer, M. Huisman, F. Kordon, and B. Steffen, Eds. Cham: Springer International Publishing, 2019, pp. 244–249.

[18] P. Havlak, “Nesting of reducible and irreducible loops,” *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 4, 1997. [Online]. Available: https://doi.org/10.1145/262004.262005

[19] A. Rossberg, “WebAssembly Core Specification,” W3C, Tech. Rep., 2019, https://webassembly.github.io/spec/core/_download/WebAssembly.pdf. [Online]. Available: https://www.w3.org/TR/wasm-core-1/

[20] G. D. Plotkin, “A Structural Approach to Operational Semantics,” 1981.

[21] A. Zakai, “Emscripten,” in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion - SPLASH '11*. ACM Press, 2011. [Online]. Available: https://doi.org/10.1145%2F2048147.2048224

[22] Z. Lin, L. Fei, G. Shuitao, Q. Xiaojun, and H. Wenbao, “Symbolic execution of network software based on unit testing,” in *2014 9th IEEE International Conference on Networking, Architecture, and Storage*, 2014, pp. 128–132.

[23] H. Cui, G. Hu, J. Wu, and J. Yang, “Verifying systems rules using rule-directed symbolic execution,” *SIGARCH Comput. Archit. News*, vol. 41, no. 1, p. 329–342, Mar. 2013. [Online]. Available: https://doi.org/10.1145/2490301.2451152

[24] K. Cong, F. Xie, and L. Lei, “Symbolic execution of virtual devices,” in *2013 13th International Conference on Quality Software*, July 2013, pp. 1–10.

[25] T. Chen, X.-s. Zhang, R.-d. Chen, B. Yang, and Y. Bai, “Conpy: Concolic execution engine for python applications,” in *Algorithms and Architectures for Parallel Processing*, X.-h. Sun, W. Qu, I. Stojmenovic, W. Zhou, Z. Li, H. Guo, G. Min, T. Yang, Y. Wu, and L. Liu, Eds. Cham: Springer International Publishing, 2014, pp. 150–163.

[26] J. F. Santos, P. Maksimović, T. Grohens, J. Dolby, and P. Gardner, “Symbolic execution for JavaScript,” in *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*, 2018.

[27] J. F. Santos, P. Maksimović, G. Sampaio, and P. Gardner, “JaVerT 2.0: compositional symbolic execution for JavaScript,” *Proceedings of the ACM on Programming Languages*, 2019.

[28] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song, “A symbolic execution framework for JavaScript,” in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, 2010.

[29] G. Li, E. Andreasen, and I. Ghosh, “SymJS: Automatic symbolic testing of JavaScript web applications,” in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[30] K. Sen, G. Necula, L. Gong, and W. Choi, “MultiSE: Multi-path symbolic execution using value summaries,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[31] J. C. King, “Symbolic execution and program testing,” *Commun. ACM*, vol. 19, no. 7, p. 385–394, Jul. 1976. [Online]. Available: https://doi.org/10.1145/360248.360252

[32] C. S. Păsăreanu and N. Rungta, “Symbolic PathFinder: Symbolic execution of Java bytecode,” in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2010.

[33] S. Khurshid, C. S. Păsăreanu, and W. Visser, “Generalized symbolic execution for model checking and testing,” in *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

[34] E. Torlak and R. Bodik, “A lightweight symbolic virtual machine for solver-aided host languages,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014.

[35] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, “Unleashing mayhem on binary code,” in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[36] R. Baldoni, E. Coppa, D. C. D’elia, C. Demetrescu, and I. Finocchi, “A survey of symbolic execution techniques,” *ACM Comput. Surv.*, 2018.

[37] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser, “Symbolic execution for software testing in practice: preliminary assessment,” in *2011 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 1066–1071.

[38] C. Cadar and K. Sen, “Symbolic execution for software testing: Three decades later,” *Commun. ACM*, 2013.

[39] S. Person, G. Yang, N. Rungta, and S. Khurshid, “Directed incremental symbolic execution,” in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2011.

[40] J. Geldenhuys, M. B. Dwyer, and W. Visser, “Probabilistic symbolic execution,” in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012.

[41] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha, “Towards automatic generation of vulnerability-based signatures,” in *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 2006. [Online]. Available: https://doi.org/10.1109%2Fsp.2006.41

[42] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, “Exe: Automatically generating inputs of death,” *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, Dec. 2008. [Online]. Available: https://doi.org/10.1145/1455518.1455522

[43] R. D. Tennent and D. R. Ghica, “Abstract models of storage,” *Higher-Order and Symbolic Computation*, vol. 13, no. 1, Apr 2000. [Online]. Available: https://doi.org/10.1023/A:1010022312623

[44] X. Leroy and S. Blazy, “Formal verification of a c-like memory model and its uses for verifying program transformations,” *Journal of Automated Reasoning*, mar 2008. [Online]. Available: https://doi.org/10.1007%2Fs10817-008-9099-0

[45] R. Bornat, “Proving pointer programs in hoare logic,” in *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2000. [Online]. Available: https://doi.org/10.1007%2F10722010_8

[46] F. Mehta and T. Nipkow, “Proving pointer programs in higher-order logic,” *Information and Computation*, 2005, 19th International Conference on Automated Deduction (CADE-19). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0890540104001804

[47] J.-C. Filliâtre and C. Marché, “Multi-prover verification of c programs,” in *Formal Methods and Software Engineering*. Springer Berlin Heidelberg, 2004. [Online]. Available: https://doi.org/10.1007%2F978-3-540-30482-1_10

[48] M. Norrish, “C formalised in hol,” 1998.

[49] H. Tuch, G. Klein, and M. Norrish, “Types, bytes, and separation logic,” in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’07. New York, NY, USA: Association for Computing Machinery, 2007. [Online]. Available: https://doi.org/10.1145/1190216.1190234

[50] F. Besson, S. Blazy, and P. Wilke, “CompCertS: A memory-aware verified c compiler using pointer as integer semantics,” in *Interactive Theorem Proving*. Springer International Publishing, 2017. [Online]. Available: https://doi.org/10.1007%2F978-3-319-66107-0_6

[51] P. Ventuzelo. Octopus: Security Analysis tool for WebAssembly module (wasm) and Blockchain Smart Contracts (BTC/ETH/NEO/EOS) . Accessed: 14th-October-2021. [Online]. Available: https://github.com/pventuzelo/octopus

[52] N. He, R. Zhang, L. Wu, H. Wang, X. Luo, Y. Guo, T. Yu, and X. Jiang, “Security analysis of eosio smart contracts,” 2020.

[53] R. M. Tsoupidi, M. Balliu, and B. Baudry, “Vivienne: Relational verification of cryptographic implementations in webassembly,” 2021.

[54] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and*

Analysis of Systems, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, p. 337–340.

[55] C. Costa, "Concolic execution for webassembly," Master's thesis, Instituto Superior Técnico, https://fenix.tecnico.ulisboa.pt/cursos/meic-a/dissertacao/846778572212567, 11 2020.

[56] F. Ramos, "Symbolic runtime modeling for automatic exploit generation," Master's thesis, Instituto Superior Técnico, https://fenix.tecnico.ulisboa.pt/cursos/meic-a/dissertacao/1128253548922710, 10 2021.

[57] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. USA: USENIX Association, 2012, p. 28.

[58] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[59] D. Beyer and T. Lemberger, "Testcov: Robust test-suite execution and coverage measurement," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1074–1077.

[60] D. Kroening and M. Tautschnig, "Cbmc – c bounded model checker," in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 389–391.

[61] i-net software. Java bytecode to WebAssembly compiler . Accessed 27th-October-2021. [Online]. Available: https://github.com/i-net-software/JWebAssembly

[62] rust lang. Empowering everyone to build reliable and efficient software. Accessed 27th-October-2021. [Online]. Available: https://github.com/rust-lang/rust