WASP: WebAssembly Symbolic Processor

Filipe Marques[1,2], José Fragoso Santos[1,2], Nuno Santos[1,2], and Pedro Adão[2,3]

[1] INESC-ID, Lisbon, Portugal
[2] Instituto Superior Técnico, Lisbon, Portugal
[3] Instituto de Telecomunicações, Portugal

Abstract. The WebAssembly Symbolic Processor, WASP, is 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 uses concolic execution to explore every feasible path of a program (up to a bound). We developed WASP-C, a symbolic framework to test C programs using WASP. WASP-C was tested against the Test-Comp 2021 benchmark suite, obtaining results comparable to well-established symbolic execution and testing tools for C.

Keywords: Concolic Testing · WebAssembly · Test-Generation · Testing C Programs

1 Introduction

The WebAssembly Symbolic Processor, WASP, is a novel concolic execution engine for testing Wasm (version 1.0) modules. WASP follows the so-called *concolic discipline* [8,23], combining concrete execution with symbolic execution and exploring one execution path at a time. However, unlike most concolic execution engines [24,27,23,22], which are implemented via program instrumentation, we implement WASP by instrumenting the Wasm reference interpreter developed by Haas et al. [9]. To this end, we lift the authors' reference interpreter from concrete values to pairs of concrete and symbolic values. By moving the instrumentation to the interpreter level, we open up the possibility for a range of optimisations in the context of concolic execution, such as application of algebraic simplification to byte-level symbolic expressions and shortcut restarts for failed assumption statements [16].

While our first goal is for WASP to be able to analyse stand-alone Wasm modules, we also aim for it to be used as a common platform for building symbolic analyses for high-level programming languages that compile to Wasm. In order to demonstrate the viability of this approach, we use it to build WASP-C, a new symbolic execution framework for testing C programs. WASP-C shows that, with a relatively small effort ($\approx$ 800 LoC), we were able to build a new concolic engine for C that is able to analyse industry-grade code and obtain results comparable to well-established symbolic execution and testing tools for C, such as KLEE [3] and VeriFuzz [2].

2 Our Approach

In this section we explore the theoretical underpinnings of the technique used by WASP and compre it with existing symbolic analysis tools for WebAssembly.

Symbolic Execution. Symbolic execution has been extensively used to find crucial errors and vulnerabilities in a broad spectrum of programming languages, such as C [8], C++ [3], Java [23], and Python [7]. Regarding the Web, there are several state-of-the-art tools for symbolically executing JavaScript code [19,20,21,15,24], demonstrating the need for such tools for the validation and testing of modern Web applications.

Symbolic execution tools can be divided into two main classes: *static* and *dynamic/concolic* [1]. Static symbolic execution engines, such as [13,18,12,25,19,20], explore the entire symbolic execution tree up to a pre-established depth, while concolic execution engines, such as [3,8,23,6,21,15,24], usually work by pairing up a concrete execution with a symbolic execution and exploring one execution path at a time. An advantage of concolic execution over static symbolic execution is that concolic execution requires less frequent interactions with the solver and a simpler memory model. There is a vast body of research on both static and concolic symbolic execution tools for a wide variety of programming languages, see [1,4,5] for comprehensive surveys on the topic. In the following, we give a detailed account of the only two existing symbolic execution tools for Wasm other than WASP.

WANA [26] is a cross-platform smart contract vulnerability detection tool employed to find vulnerabilities in EOSIO [14] and Ethereum smart contracts [11]. WANA is based on static symbolic execution and operates over Wasm bytecode. To detect vulnerabilities in smart contracts, WANA comes with three heuristics for EOSIO smart contracts and four for Ethereum smart contracts. Unlike WASP, WANA lacks a stand-alone symbolic execution engine for Wasm. Hence, it is not possible to run WANA on arbitrary Wasm code without refactoring its internal architecture. For this reason, we were unable to evaluate WANA and compare its performance with those of WASP and Manticore.

Manticore [17] is a symbolic execution framework for binaries and smart contracts. Manticore is highly flexible, supporting a wide range of binaries and computing environments, including Wasm bytecode. When it comes to Wasm, Manticore does not expose dedicated primitives for constructing and reasoning over symbolic values at the source language level. Symbolic inputs and constraints are created as part of a complex Python script that must be written for each test [10], which initialises the symbolic state and calls the appropriate Wasm module. This process does not scale for a broad evaluation, as one would have to manually write a python script for each symbolic test. Nonetheless, we compare the performance of WASP with that of Manticore [17] in the analysis of a stand-alone Wasm implementation of a B-tree data structure, demonstrating that WASP is consistently faster.

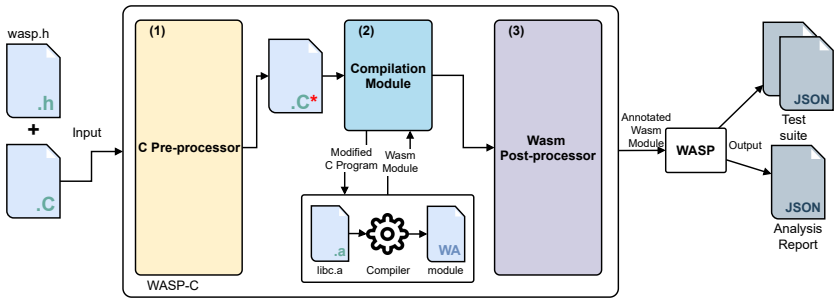

Fig. 1. High-level architecture of WASP-C.

3 Tool Architecture

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. Since WASP does not directly operate over the C source code, WASP-C is comprised of three modules whose end goal is to generate a Wasm program for WASP to analyse.

WASP-C is implemented in python and is composed of three essential modules: a *C Pre-processor*, a *Compilation Module*, and a *Wasm Post-processor*, which interact with each other according to the high-level architecture described in Figure 1. Using WASP as a submodule, WASP-C concolically executes C programs as follows. First, the *C Pre-processor* parses the given program using a standard C parser called *pycparser*,[4] generating an abstract syntax tree (AST) that is then sent to a specialised C visitor (step 1). Our specialised C visitor traverses the AST, replacing binary operators such as logical ANDs and ORs with specific function calls to avoid spurious branching. Then the AST is exported back to a C program, which is subsequently compiled into Wasm by the *Compilation Module* (step 2). Lastly, the *Wasm Post-processor* processes the obtained Wasm module so as to inject the appropriate WASP symbolic primitives (step 3).

4 Strengths and Weaknesses

The main strength of WASP is that it can quickly generate inputs that trigger assertion violations in the program. For WASP-C's evaluation we tested WASP-C against the benchmark suite of Test-Comp 2021. For the `cover-error` meta-category WASP-C was able to score 360 points which would have yield a third-place in Test-Comp 2021.

[4] `https://github.com/eliben/pycparser`

WASP only implements standard branch search policies, such as DFS, BFS, and random selection; therefore, WASP has difficulties in generating high-coverage test suites.

5 Tool Configuration and Setup

WASP and WASP-C are publicly available at the URL `https://github.com/wasp-platform/wasp`. Its Test-Comp 2023 variant is available at the URL `blahbleh`. The benchmark description file is `wasp-c.xml`. To install and run the tool, follow the instructions provided in `README.md`. A sample run command is as follows:

```
./bin/wasp-c --smt-assume --policy breadth --test-comp -p cover-error.prp
↪  examples/CostasArray-10.c
```

6 Software Project and Contributors

WASP and WASP-C is developed by the authors and their students. We would also like thank Carolina Costa with whom we designed a preliminary version of WASP.

References

1. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Computing Surveys (2018)
2. Basak Chowdhury, A., Medicherla, R.K., R, V.: VeriFuzz: Program Aware Fuzzing. In: Tools and Algorithms for the Construction and Analysis of Systems (2019)
3. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In: USENIX Conference on Operating Systems Design and Implementation (2008)
4. Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C.S., Sen, K., Tillmann, N., Visser, W.: Symbolic execution for software testing in practice: preliminary assessment. In: International Conference on Software Engineering (2011)
5. Cadar, C., Sen, K.: Symbolic execution for software testing: Three decades later. Communications of the ACM (2013)
6. Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D.: Unleashing Mayhem on Binary Code. In: IEEE Symposium on Security and Privacy (2012)
7. Chen, T., Zhang, X.s., Chen, R.d., Yang, B., Bai, Y.: Conpy: Concolic execution engine for python applications. In: Algorithms and Architectures for Parallel Processing (2014)
8. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed automated random testing. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (2005)
9. Haas, A., Rossberg, A., Schuff, D.L., Titzer, B.L., Holman, M., Gohman, D., Wagner, L., Zakai, A., Bastien, J.: Bringing the web up to speed with WebAssembly. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (2017)

10. Hennenfent, E.: Symbolically Executing WebAssembly in Manticore, https://blog.trailofbits.com/2020/01/31/symbolically-executing-webassembly-in-manticore, accessed: 30th-November-2021
11. Kashyap, B.: Introduction to smart contracts, https://ethereum.org/en/developers/docs/smart-contracts, accessed: 30th-November-2021
12. Khurshid, S., Păsăreanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Tools and Algorithms for the Construction and Analysis of Systems (2003)
13. King, J.C.: Symbolic execution and program testing. Communications of the ACM (1976)
14. Larimer, D., Blumer, B.: EOS.IO Technical White Paper, https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md, accessed: 30th-November-2021
15. Li, G., Andreasen, E., Ghosh, I.: SymJS: Automatic symbolic testing of JavaScript web applications. In: ACM SIGSOFT International Symposium on Foundations of Software Engineering (2014)
16. Marques, F., Fragoso Santos, J., Santos, N., Adão, P.: Concolic Execution for WebAssembly. In: European Conference on Object-Oriented Programming (2022)
17. Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., Dinaburg, A.: Manticore: A user-friendly symbolic execution framework for binaries and smart contracts (2019)
18. Păsăreanu, C.S., Rungta, N.: Symbolic PathFinder: Symbolic Execution of Java Bytecode. In: IEEE/ACM International Conference on Automated Software Engineering (2010)
19. Santos, J.F., Maksimović, P., Grohens, T., Dolby, J., Gardner, P.: Symbolic execution for JavaScript. In: International Symposium on Principles and Practice of Declarative Programming (2018)
20. Santos, J.F., Maksimović, P., Sampaio, G., Gardner, P.: JaVerT 2.0: compositional symbolic execution for JavaScript. Proceedings of the ACM on Programming Languages (2019)
21. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: IEEE Symposium on Security and Privacy (2010)
22. Sen, K., Agha, G.: CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In: CAV (2006)
23. Sen, K., Marinov, D., Agha, G.: CUTE: A Concolic Unit Testing Engine for C. ACM SIGSOFT Software Engineering Notes (2005)
24. Sen, K., Necula, G., Gong, L., Choi, W.: MultiSE: Multi-path symbolic execution using value summaries. In: Joint Meeting on Foundations of Software Engineering (2015)
25. Torlak, E., Bodik, R.: A lightweight symbolic virtual machine for solver-aided host languages. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (2014)
26. Wang, D., Jiang, B., Chan, W.K.: WANA: Symbolic Execution of Wasm Bytecode for Cross-Platform Smart Contract Vulnerability Detection (2020)
27. You, S., Findler, R.B., Dimoulas, C.: Sound and complete concolic testing for higher-order functions. In: ESOP (2021)