WasmDyPa: Effectively Detecting WebAssembly Bugs via Dynamic Program Analysis

Wenlong Zheng, Baojian Hua*, and Zhuochen Jiang
School of Software Engineering, University of Science and Technology of China, China
Suzhou Institute for Advanced Research, University of Science and Technology of China, China
zwl21@mail.ustc.edu.cn, bjhua@ustc.edu.cn*, jzc666@mail.ustc.edu.cn
* Corresponding author.

Abstract—Safe binary execution is a crucial requirement in today's security-critical computing infrastructures. WebAssembly is an emerging language designed for safe binary execution that has been deployed in many security-critical domains, such as blockchain, edge computing, and clouds. However, WebAssembly's security guarantee is not a panacea, and recent studies have revealed a large spectrum of security issues such as integer overflows and memory vulnerabilities, leading to serious security hazards to WebAssembly applications. In this paper, we present WasmDyPa, the first automated bug detection framework for WebAssembly programs based on dynamic program analysis with three primary components: 1) an input generator for WebAssembly binaries; 2) static instrumentation hooks with extensible interfaces to record runtime information; and 3) dynamic program analysis algorithms as security plugins to detect vulnerabilities. We have implemented a software prototype for WasmDyPa, and have conducted experiments to evaluate the effectiveness, usefulness, performance of our approach. The experimental results demonstrated that WasmDyPa can accurately detect vulnerabilities with an 88.24% precision and a 93.75% recall. Furthermore, WasmDyPa detected 56 bugs in real-world WebAssembly programs, including 2 integer overflows and 54 memory bugs.

Keywords–WebAssembly, Security, Dynamic analysis

1. Introduction

Today's cloud or edge computing infrastructures put forward higher requirements for the safe execution of binary programs on them. WebAssembly (Wasm) [1] is an emerging portable binary distribution format that allows for safe program execution with near-native execution efficiency. Due to Wasm's technical advantages of type safety and intra-process lightweight sandboxing [2], Wasm has been extensively used in a large spectrum of safety-critical scenarios [3] [4].

While Wasm makes a significant step toward defining a secure binary format, existing studies [5] [6] have revealed that Wasm programs are still vulnerable and exploitable due to two main root causes. First, the limitations of Wasm's type system, such as missing checks for value and its propagation can lead to undetected overflows and even buffer overflows in memory allocation. Second, Wasm relies on manual memory management which can lead to notorious memory vulnerabilities such as double-free or use-after-free. Despite this urgent need, existing studies and tools [7] for Wasm cannot detect these vulnerabilities systematically.

To address these limitations, this paper presents WasmDyPa, a fully automated bug detection framework for Wasm programs based on dynamic analysis in conjunction with static instrumentation, leveraging runtime information that is hard to obtain in static analysis. WasmDyPa consists of three main components: 1) a test case generator to generate inputs for Wasm programs; 2) static instrumentation hooks to collect runtime information of susceptible Wasm instructions and 3) dedicated analysis algorithms as plugins to detect vulnerabilities based on the runtime information collected by hooks.

To validate our design and implementation, we conducted a systematic evaluation of WasmDyPa on micro- and real-world benchmarks. Experiment results showed that WasmDyPa is effective in achieving an 88.24% precision and a 93.75% recall on the micro-benchmark with acceptable overhead. Moreover, WasmDyPa is useful for detecting vulnerabilities including 2 integer overflows and 54 memory bugs from the real-world benchmark with 941 Wasm programs.

2. WasmDyPa Approach

This section discusses our approach by presenting the design and implementation of WasmDyPa.

2.1 Design.

We present, in Figure 1, the overall architecture of WasmDyPa, consisting of three primary modules, which will be discussed next in detail, respectively.

Input Generation. Our benchmark contains Wasm programs that require inputs, and the generator is designed for this. However current Wasm fuzzing tools mainly focus on the execution paths coverage from the static aspect, but not capable enough of generating malicious inputs for Wasm programs to trigger potential bugs during runtime. For example, tools may generate a harmless input to cover a path with no overflown handler on mathematic operations, which can invade the dynamic detection. Hence, the generator instruments preconditions to augment inputs for further dynamic analysis.

Hook Instrumentation and Validation. The hook instrumentation takes as input a Wasm program and outputs the instrumented Wasm by placing hooks before and after corresponding instructions to collect specific runtime information

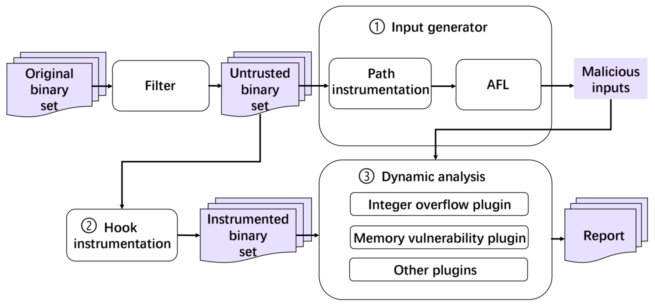

Figure 1: The architecture of WASMDYPA.

for subsequent analysis. One key design point is the semantic consistency after the instrumentation. Hence, the validation module takes in both the original and instrumented Wasm programs to guarantee the Wasm syntax and semantic consistency based on a thorough analysis of the behavior of programs.

Dynamic Analysis Algorithms as Security Plugins. The security plugins provide an extendable and general template for dynamic analysis algorithms to detect potential vulnerabilities. The plugins for algorithms takes in the runtime information collected and transferred to the template's interfaces by hooks, and detect the runtime abnormal behavior based on the specific information such as memory address, and the computation and propagation of variable values.

2.2 Implementation.

We have implemented a software prototype, following the architecture of WASMDYPA. First, we have implemented the input generator as a compiler rewriting pass to instrument precondition predicates on the abstract syntax trees offered by Fuzzm. Second, we have ported the Web-confined hook infrastructure Wasabi to a standalone Wasm virtual machine *i.e.,* Wasm-Micro-Runtime (WAMR) to make WASMDYPA versatile in executing arbitrary scenarios of Wasm applications. Meanwhile, to validate the instrumented Wasm, we have leveraged the `wasm-validate` facility from the WABT, offering well-formedness guarantees and consistency of instrumented Wasm code. Finally, we have provided a standard template to write detection algorithms as security plugins and two well-rounded and straightforward plugins for evaluation, which can be dynamically linked to WAMR.

3. EVALUATION

The WASMDYPA is still under heavy development, and we have conducted some initial experiments with it. First, we curated two benchmarks: 1) micro-benchmark and 2) real-world benchmark with 941 Wasm programs (61 from CWE, and 880 Wasm binaries) to conduct our study. Second, to evaluate the effectiveness of WASMDYPA, we applied WASMDYPA to the micro-benchmark, and attached analysis hooks on susceptive sites to carry out dynamic analysis. Experimental results showed that WASMDYPA achieves an 88.24% precision and a 93.75% recall with an acceptable analysis time under 0.135 seconds. Finally, to evaluate the usefulness of WASMDYPA, we applied it to our curated real-world benchmark and detected 56 bugs, including 2 integer overflow bugs and 54 memory bugs. The survey results demonstrated that WASMDYPA is capable of detecting vulnerabilities in Wasm.

4. RELATED WORK

Dynamic analysis. Dynamic analysis has been extensively studied. Agrawal et al. [8] proposed dynamic slicing as a complement to static slicing [9]. Newsome et al. [10] proposed a dynamic taint analysis featuring attack detection. However, these works cannot apply to Wasm due to feature discrepancies. On the contrary, our work can detect vulnerabilities in Wasm programs from both non-Web and Web scenarios.

5. SUMMARY

This paper presented WASMDYPA, the first infrastructure to detect Wasm bugs by dynamic program analysis. WASMDYPA consists of input generation, hook instrumentation, and dynamic analysis algorithms as security plugins. The work in this paper will make Wasm not only an efficient but also a safer programming language.

REFERENCES

[1] "Webassembly," https://webassembly.org/.
[2] B. McFadden, T. Lukasiewicz, J. Dileo, and Engler, "Security chasms of wasm," Tech. Rep., Aug. 2018.
[3] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "Ct-wasm: Type-driven secure cryptography for the web ecosystem," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, Jan. 2019.
[4] "Apache/apisix: The cloud-native api gateway," https://github.com/apache/apisix.
[5] D. Lehmann, J. Kinder, and M. Pradel, "Everything old is new again: Binary security of webassembly," in *USENIX Security*, 2020, p. 19.
[6] A. Hilbig, D. Lehmann, and M. Pradel, "An empirical study of real-world webassembly binaries: Security, languages, use cases," in *Proceedings of the Web Conference 2021*. Ljubljana Slovenia: ACM, Apr. 2021, pp. 2696–2708.
[7] D. Lehmann and M. Pradel, "Wasabi: A framework for dynamically analyzing webassembly," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. Providence RI USA: ACM, Apr. 2019, pp. 1045–1058.
[8] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *ACM SIGPLAN Notices*, vol. 25, no. 6, pp. 246–256, Jun. 1990.
[9] M. Harman and R. Hierons, "An overview of program slicing," *Software Focus*, vol. 2, no. 3, pp. 85–92, 2001.
[10] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," *NDSS*, vol. 5, pp. 3–4, 2005.