SWFuzz: Structure-sensitive Wasm Fuzz

Ziyi Guo, Jiashui Wang, Yan Chen

Background

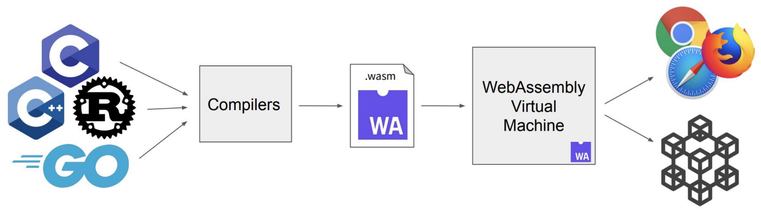
- WebAssembly(WASM) has become a critical component in real-world systems, including Browsers, Blockchains and so on.
  - WASM: A portable binary-code format and a corresponding text format for executable programs as well as software interfaces for facilitating interactions between such programs and their host environment.
- Fuzzing Techniques are of great effectiveness to find security vulnerabilities in real-world environment.
  - Coverage-Guided Fuzzing.
  - Different mutators to fit the fuzzing targets.

Research Objectives

- A deeper understanding of real-world WASM attacking surfaces.
- A comprehensive evaluation of real-world WASM security issues.
- Design a fuzzer of WASM, combining the attacking surfaces.
- Evaluate the fuzzer under different real-world environments.

WebAssembly Attacking Surfaces

- **Comprehensive Attacking Surfaces**[1]: WebAssembly is a binary format, to execute codes based on this format, the virual machine is introduced. However, in real-world, WASM virual machine has multiple design and implementation, in order to match different environments. The critical attack surfaces of it, indicate to the **instruction execution(IE)** and **binary format parser(BFP)**.

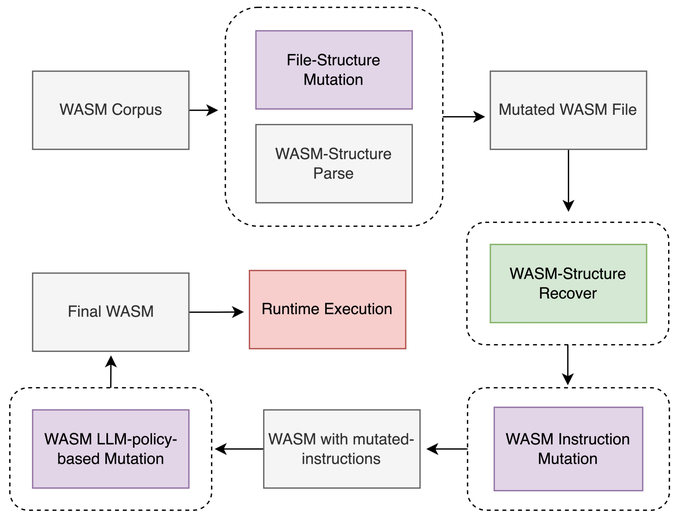


[1] Ventuzelo P, et al. A Journey Into Fuzzing WebAssembly Virtual Machines[C]. Blackhat USA, 2022.

Method: Binary Structure-sensitive Mutation

- We design a binary level structure-sensitive mutation method to perform the effectively fuzzing process.
  - **Header Rebuild.** Parsing not only the headers but also delving into each section to extract both control information and associated data.
  - **Mutation.** It applies a granular approach to each section's data and code. By segmenting the sections further into nodes, the framework achieves a finer understanding and control over the binary, setting the stage for targeted byte mutations.
  - **Structure Fix.** The Structure Fix stage is designed to remedy these issues by repairing the mutated file to a state that is both structurally sound and executable within a WASM runtime environment, without diluting the efficacy of the introduced mutations. This involves two primary tasks: Length Adjustment and Dependency and Data-Flow Correction.

Full Design



Method: Code Level Mutation

Mutate the codes to achieve better coverage and execution path.

- Effective mutation for instruction.
- We define different mutation strategies for instruction level mutation.

Prototype	Mutation
Functions	Insert/Remove/Move...
Export/Import	Insert/Remove/Move...
Variables	Insert/Remove/Move...

Examples:

```
if (module.Funclist.size() == 0) {
    return module;
}
Instruction opcode = RandomInst();
(Args, bool) (state, success) = GetRandomArgs(opcode);
if (success == false) {
    return module;
}
Instruction* expr = Generator(opcode, state);
Function func = module.Funclist.RandomElem();
func.InstList.RandomInsert(expr);
return module;
```

```
Typelist funcTypeVector = module.Typelist.select(type=Func);
if (funcTypeVector.size() == 0) {
    return module;
}
Type funcType = funcTypeVector.RandomElem();
Function* func = new Function(funcType);
module.FunctionElem.AddElem(func);
return module;
```

Preliminary Results

- We evaluate our fuzzing system on different current WASM runtimes, including **wasm-micro-runtime/wasm3/wac/vmir**
- We have received multiple CVEs: **CVE-2024-25431, CVE-2024-27527, CVE-2024-27528, CVE-2024-27529, CVE-2024-27530, CVE-2024-27532**
- We conduct the systematically analysis to current bugs, and confirm the effectiveness of structure mutation.

Expected Results

- We will systematically evaluate our fuzzing system in coverage level, compared to traditional AFL/AFL++ and etc.
- We will help the community to fix these vulnerabilties, to improve the reliability of real-world lower-level infra.