A Journey Into Fuzzing WebAssembly Virtual Machines

Patrick Ventuzelo
Patrick Ventuzelo (@Pat_Ventuzelo)

- Founder & CEO of FuzzingLabs | Senior Security Researcher
  - Fuzzing and vulnerability research
  - Development of security tools

- Training/Online courses
  - Rust Security Audit & Fuzzing
  - Go Security Audit & Fuzzing
  - WebAssembly Reversing & Analysis
  - Practical Web Browser Fuzzing

- Main focus
  - Fuzzing, Vulnerability research
  - Rust, Golang, WebAssembly, Browsers
  - Blockchain Security, Smart contracts

- Previously speaker at:
  - OffensiveCon, REcon, RingZer0, ToorCon, hack.lu, NorthSec, FIRST, etc.
Introduction to WebAssembly
What is **WebAssembly**?

- **Binary** instruction format for a **stack-based virtual machine**
  - Low-level bytecode
  - Compilation target for C/C++/Rust/Go/etc.

- Generic evolution of [NaCl & Asm.js](https://asmjs.org)
- [W3C](https://www.w3.org) standard
- MVP 1.0 (March 2017), MVP 2.0 ([2022/2023](https://www.webkit.org/blog/webassembly-mvp-2-0/))
- Natively supported in all major browsers

- WebAssembly goals:
  - Be fast, **efficient**, and portable
  - Easily readable and **debuggable**
  - **Safe** (using sandboxed execution environment)
  - Open and **modulable**
How WebAssembly works?

Compilers → .wasm → WebAssembly Virtual Machine

#BHUSA @BlackHatEvents
Step 1: Compilation into WebAssembly module

Source code & Compiler toolchains

Compilers

LLVM, Emscripten, Binaryen

.Wasm

WebAssembly Virtual Machine
WebAssembly Binary Format

C/C++

```c
int fib(int n)
{
    if (n == 0 || n == 1)
        return n;
    else
        return (fib(n-1) + fib(n-2));
}
```

Rust

```rust
fn fib(n: u32) -> u32 {
    match n {
        0 => 1,
        1 => 1,
        _ => fib(n - 1) + fib(n - 2),
    }
}
```

Compilation

binary file (.wasm)

```
0061 736d 0100 0000 0186 8080 8080 0010 0000 0017f 017f 0382 8080 8000 0100 0484 8080 8000 0170 0000 0583 8080 8000 0100 0106 8180 8080 0000 0790 8080 8000 0206 6d65 6d6f 7279 0200 0366 6962 0000 0aa7 8080 8000 01a1 8080 8000 0002 4020 0041 0172 4101 470d 0020 000f 0b20 0041 7f6a 1000 2000 417e 6a10 006a 0b
```
Step 2: Execution by the WebAssembly VM

Source code & Compiler toolchains
- C, C++, R
- Compilers
- LLVM, Emscripten, Binaryen

Runtime & Host environments
- WebAssembly Virtual Machine
  - V8, wasmer, wasmtime

Compilers
WebAssembly VM - Execution stages
1. **Decoding/Parsing:** The binary format is parsed and converted into a module.
1. **Decoding/Parsing**: The binary format is parsed and converted into a module
2. **Validation**: The decoded module undergoes validation checks (such as type checking)
1. **Decoding/Parsing:** The binary format is parsed and converted into a module
2. **Validation:** The decoded module undergoes validation checks (such as type checking)
3. **Instantiation:** Creation of a module instance with all the context instantiated
WebAssembly VM - Instantiation

Host (OS, Browser) - Shared
wasm instance (VM) - immutable

memories
globals
Tables

Functions
0 1 2 3

Indirect function table
3 0 1 1 2

Execution stack
1. **Decoding/Parsing**: The binary format is parsed and converted into a module
2. **Validation**: The decoded module undergoes validation checks (such as type checking)
3. **Instantiation**: Creation of a module instance with all the context instantiated
4. **Execution/Invocation**: Exported functions are called by the host over the module instance
Step 2: Execution by the WebAssembly VM

**Source code & Compiler toolchains**

- Compilers
  - LLVM, Emscripten, Binaryen

**Runtime & Host environments**

- WebAssembly Virtual Machine
  - V8, wasmer, wasmtime
WebAssembly VM - Use-cases

- **Standalone VM** (server)
  - Edge computing
  - Back-end apps
    - Nodejs
  - Mobile & Desktop apps
  - IoT & Embedded OS
  - Blockchain
    - Polkadot, Substrate, Cosmos, NEAR
    - Spacemesh, Golem, EOS, DFINITY

- **Browser** (client)
  - Video, Audio & Image processing
  - Videos Games
  - Complexe web apps
    - Autocad, Google Earth
    - Photoshop, Shopify, Figma
  - OS Emulation
Focus of this talk: WebAssembly VM

Source code & Compiler toolchains

Runtime & Host environments

Compilers

.Wasm

WebAssembly Virtual Machine

#BHUSA @BlackHatEvents
Goal: Find bugs on every stage on different VMs!

1. **Decoding/Parsing**: The binary format is parsed and converted into a module
2. **Validation**: The decoded module undergoes validation checks (such as type checking)
3. **Instantiation**: Creation of a module instance with all the context instantiated
4. **Execution/Invocation**: Exported functions are called by the host over the module instance
1. Coverage-guided fuzzing
Fuzzing strategy: Coverage-guided fuzzing

- **Coverage-guided** fuzzing
  - Observe how inputs are processed to **learn** which mutations are interesting.
  - Save inputs to be **re-used** and mutated in future iterations.
Input: WebAssembly Binary Format

- Module structure
  - **Header**: magic number + version
  - 11 **Sections**: may appear at most once
  - 1 **custom** section: unlimited
Targets: Standalone VMs & parsing libraries

- Targets (C/C++)
  - **Binaryen**: Compiler and toolchain libraries
  - **WABT**: The WebAssembly Binary Toolkit
  - **Wasm3**: WebAssembly interpreter
  - **WAMR**: WebAssembly Micro Runtime
  - **WAC**: WebAssembly interpreter in C
  - **Radare2**: Reverse engineering framework
  - Etc.
C/C++ Coverage-guided Fuzzing

- C/C++ Fuzzers
  - **AFL**: american fuzzy lop
  - **Honggfuzz**: Feedback-driven/evolutionary fuzzer
  - **AFL++**: AFL with community patches

- Complexity: **None**
  - Instrumentation using custom gcc/clang
  - Overwrite CC or CXX flags
  - Preferred AFL++ instead of vanilla AFL

---

**American Fuzzy Lop 2.52b (iwasm)**

<table>
<thead>
<tr>
<th>process timing</th>
<th>run time:</th>
<th>3 days, 16 hrs, 50 min, 25 sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>last new path</td>
<td>0 days, 0 hrs, 0 min, 37 sec</td>
<td></td>
</tr>
<tr>
<td>last uniq crash</td>
<td>0 days, 0 hrs, 4 min, 33 sec</td>
<td></td>
</tr>
<tr>
<td>last uniq hang</td>
<td>0 days, 9 hrs, 40 min, 58 sec</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle progress</th>
<th>now processing</th>
<th>28.8k* (78.69%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>paths timed out</td>
<td>0 (0.00%)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stage progress</th>
<th>now trying:</th>
<th>arith 8/8</th>
</tr>
</thead>
<tbody>
<tr>
<td>stage execs</td>
<td>42.1k/64.9k (65.79%)</td>
<td></td>
</tr>
<tr>
<td>total execs</td>
<td>245M</td>
<td></td>
</tr>
<tr>
<td>exec speed</td>
<td>293.5/sec</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>map coverage</th>
<th>map density:</th>
<th>2.29% / 24.59%</th>
</tr>
</thead>
<tbody>
<tr>
<td>count coverage</td>
<td>2.28 bits/tuple</td>
<td></td>
</tr>
<tr>
<td>findings in depth</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>overall results</th>
<th>cycles done:</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>total paths:</td>
<td>40.7k</td>
<td></td>
</tr>
<tr>
<td>uniq crashes:</td>
<td>4999</td>
<td></td>
</tr>
<tr>
<td>uniq hangs:</td>
<td>500+</td>
<td></td>
</tr>
</tbody>
</table>

---

**American Fuzzy Lop 2.52b (war)**

<table>
<thead>
<tr>
<th>process timing</th>
<th>run time:</th>
<th>0 days, 3 hrs, 8 min, 19 sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>last new path</td>
<td>0 days, 0 hrs, 0 min, 8 sec</td>
<td></td>
</tr>
<tr>
<td>last uniq crash</td>
<td>0 days, 0 hrs, 4 min, 4 sec</td>
<td></td>
</tr>
<tr>
<td>last uniq hang</td>
<td>0 days, 0 hrs, 5 min, 27 sec</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle progress</th>
<th>now processing</th>
<th>942k (40.52%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>paths timed out</td>
<td>0 (0.00%)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>stage progress</th>
<th>now trying:</th>
<th>arith 8/8</th>
</tr>
</thead>
<tbody>
<tr>
<td>stage execs</td>
<td>2432/7857 (36.95%)</td>
<td></td>
</tr>
<tr>
<td>total execs</td>
<td>10.0M</td>
<td></td>
</tr>
<tr>
<td>exec speed</td>
<td>2041/sec</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>map coverage</th>
<th>map density:</th>
<th>0.32% / 2.47%</th>
</tr>
</thead>
<tbody>
<tr>
<td>count coverage</td>
<td>4.00 bits/tuple</td>
<td></td>
</tr>
<tr>
<td>findings in depth</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>overall results</th>
<th>cycles done:</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>total paths:</td>
<td>2325</td>
<td></td>
</tr>
<tr>
<td>uniq crashes:</td>
<td>538</td>
<td></td>
</tr>
<tr>
<td>uniq hangs:</td>
<td>77</td>
<td></td>
</tr>
</tbody>
</table>

---

#BHUSA @BlackHatEvents
Results: ~46 bugs/vulnerabilities

- **Binaryen**
  - Out-of-bound read - [issue](#)

- **WABT**
  - Assertion errors - [issue#1], [issue#2], [issue#3], [issue#4]
  - Uncontrolled memory allocation - [issue](#)

- **WAMR**
  - Null pointer dereference - [issues](#) (5)
  - Heap out of bounds read - [issues](#) (29)
  - Assertion errors - [issue#1], [issue#2]
  - Heap out of bounds write - [issue](#)
  - Segmentation fault - [issue](#)

- **Radare2**
  - Heap out of bounds read - [issue](#)
  - Heap out of bounds read - [issue](#)

---

```
Invalid read of size 8
at 0x111716E: wasm_interp_call_func_bytecode (wasm_interp.c:1180)
by 0x1100FC: wasm_interp_call_wasm (wasm_interp.c:2158)
by 0x10D446: wasm_runtime_call_wasm (wasm_runtime.c:102)
by 0x10CB41: wasm_application_execute_main (wasm_application.c:199)
by 0x10BAD7: app_instance_main (main.c:54)
by 0x10C9EA: main (main.c:217)
Address 0x18 is not stack'd, malloc'd or (recently) free'd
```

```
Program received signal SIGSEGV, Segmentation fault.
0x0000055555a9a02f in wasm::WasmBinaryBuilder::readImports() ()
(gdb) bt
#0 0x0000055555a9a02f in wasm::WasmBinaryBuilder::readImports() ()
#1 0x0000055555a9f918 in wasm::WasmBinaryBuilder::read() ()
#2 0x0000055555ac498 in wasm::ModuleReader::readBinaryData(std::vector<wasm::<...>*, void*)
#3 0x0000055555ac921 in wasm::ModuleReader::readBinary(std::vector<wasm::<...>*, void*)
#4 0x0000055555ace780 in wasm::ModuleReader::read(std::vector<wasm::<...>*, void*)
#5 0x000005555557ead2 in main ()
```

==3759== ERROR: AddressSanitizer: heap-buffer-overflow
READ of size 1 at 0x611000016380 thread T0
Fuzzing strategy: Improvements #1

- **Reusing corpora** between all targets
Fuzzing strategy: Improvements #1

- **Reusing corpora** between all targets
- Add crashing files inside the existing corpus
  - It might make crash some other targets
2. In-process fuzzing
Fuzzing strategy: In-process fuzzing

- In-Process fuzzing
  - Fuzz a specific entry point of the program in **only one dedicated process**
  - For every test case, the process isn’t restarted but the values are changed in memory.
Targets: Standalone VMs & parsing libraries

- Targets (Rust)
  - Wasmer: WebAssembly **Runtime** supporting WASI and Emscripten
  - Wasmtime: A standalone **runtime** for WebAssembly
  - wain: WebAssembly **interpreter** written in Rust from scratch
  - Wasmparser: **Decoding/parsing library** of wasm binary files
  - wasmi: WebAssembly (Wasm) **interpreter**.
  - Cranelift: JIT compiler for wasm
  - Lucet: Sandboxing WebAssembly Compiler
  - Etc.

- Targets
  - pywasm: A WebAssembly interpreter written in pure **Python**
  - webassembllyjs: JavaScript Toolchain for WebAssembly
Rust In-process fuzzing

- Rust Fuzzers
  - cargofuzz: A cargo subcommand for fuzzing with libFuzzer
  - honggfuzz-rs: Fuzz your Rust code with Honggfuzz
  - afl.rs: Fuzzing Rust code with AFLplusplus

- Complexity: Low
  - You need to write some fuzzing harnesses
  - honggfuzz-rs is my favorite (faster and better interface)
  - New fuzzer cargo-libafl is promising
Python/JS In-process fuzzing

- Fuzzers
  - **Atheris**: Coverage-guided Python fuzzing engine based on Libfuzzer
  - **jsfuzz**: Coverage-guided fuzzer for javascript/nodejs packages

- Complexity: **Low**
  - You need to write some fuzzing harnesses
  - Learn how to use different fuzzing frameworks
Results: ~62 bugs/vulnerabilities

- Results
  - Wasmer - issues (22)
  - Cranelift - issues (2)
  - Wasm/parser - issues (3)
  - Wasmtime - issues (17)
  - Wain - issues (4)
  - Lucet - issues (2)
  - Pywasm - not reported (10)
  - Webassemblyjs - issue

- Type of bugs found
  - Panicking macros
  - Index out of bound panic
  - Assertion failure
  - Unwrapping panics
  - Arithmetic overflows
  - Out of Memory (OOM) error
  - Unhandled exception (Python)
Fuzzing strategy: Improvements #2

- **Improving the corpora** by gathering valid inputs/seeds from internet
  - WebAssembly/spec: WebAssembly core testsuite
  - Existing WebAssembly fuzzing corpora - [here](#), [here](#) or [there](#)
3. Grammar-based fuzzing
Fuzzing strategy: Improvements #3

- **Add new fuzzing harnesses** to target validation entry points.
  - Module decoding will also be called by the validation function

```rust
pub fn wasmi_validate(data: &[u8]) -> bool {
    use parity_wasm::{deserialize_buffer, elements};
    use wasmi_validation::{validate_module, PlainValidator};

    let module: elements::Module = match deserialize_buffer(&data) {
        Ok(module) => module,
        _ => return false,
    };
    validate_module::<PlainValidator>(&module).is_ok()
}
```

```rust
pub fn fuzz wasmi_validate(data: &[u8]) -> bool {
    let store = Store::default();
    Module::validate(&store.engine(), &data).is_ok()
}
```

```rust
pub fn fuzz_wasmparser_validate(data: &[u8]) -> bool {
    use wasmparser::validate;
    validate(&data).is_ok()
}
```

```rust
/// Fuzzing `wasmtime::validate` with default Store/Config/Engine
pub fn fuzz wasmtime_validate(data: &[u8]) -> bool {
    let store = match get_store_all_features(Strategy::Cranefly) {
        None => return false,
        Some(a) => a,
    };
    Module::validate(&store.engine(), &data).is_ok()
}
```

```rust
pub fn fuzz_wain_validate(data: &[u8]) -> bool {
    // Parse binary into syntax tree
    match parse(&data) {
        // Validate module
        Ok(tree) => validate(&tree).is_ok(),
        Err(_) => false,
    }
}
```
Main issue: Strict module validation mechanism

- The decoded module undergoes **validation checks** (such as type checking)
  - Validation mechanism is documented in the specs ([here](#))
    - Conventions
    - Types
    - Instructions
    - Modules

The WebAssembly stack machine is restricted to structured control flow and structured use of the stack. This greatly simplifies a **one-pass verification**, avoiding a fixpoint computation like that of other stack machines such as the Java Virtual Machine (prior to stack maps). This also simplifies compilation and manipulation of WebAssembly code by other tools. Further generalization of the WebAssembly stack machine is planned post-MVP, such as the addition of multiple return values from control flow constructs and function calls.

- Different implementations
  - `wasm-validator` tool (binaryen - C/C++)
  - `wasm-validate` tool (wabt - C/C++)
  - `WebAssembly.validate` (JS API - JavaScript)

- Further reading:
  - WebAssembly Core Specification: Validation Algorithm - [link](#)
  - Mechanising and Verifying the WebAssembly Specification - [link](#)
  - “One pass verification process” explains - [link](#)
Standalone VMs: Grammar-based fuzzing

- **Grammar-based** fuzzing
  - Grammar allows for systematic and efficient test generation, particularly for complex formats.
  - Convert **WebAssembly text** files into wasm binaries and **add them to the corpora**
    - Found interesting wat files online, create and generate custom wat files
Input: WebAssembly Binary Format & Text Format

C/C++

```c
int fib(int n) {
    if (n == 0 || n == 1)
        return n;
    else
        return (fib(n-1) + fib(n-2));
}
```

Rust

```rust
fn fib(n: u32) -> u32 {
    match n {
        0 => 1,
        1 => 1,
        _ => fib(n - 1) + fib(n - 2),
    }
}
```

### Compilation

- **binary file (.wasm)**
- **wasm text format (.wat)**
Input: WebAssembly Text Format

- **Standardized text format**
  - File extensions: .wat
  - **S-expressions** (like LISP): Module and section definitions
  - **Linear representation**: Functions body and Low-level instructions

- **MVP Instruction set**
  - Small Turing-complete ISA: ~172 instructions
  - Data types: i32, i64, f32, f64
  - **Control-Flow** operators
    - Label
    - Branch
    - Function call
  - **Memory** operators
  - **Variables** operators
  - **Arithmetic** operators
  - **Constant** operators
  - **Conversion** operators
# MVP 1.0 Instruction Set Architecture (ISA)

<table>
<thead>
<tr>
<th>i32</th>
<th>i64</th>
<th>f32</th>
<th>f64</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>add</td>
<td>f32.add</td>
<td>f64.add</td>
</tr>
<tr>
<td>sub</td>
<td>sub</td>
<td>f32.sub</td>
<td>f64.sub</td>
</tr>
<tr>
<td>mul</td>
<td>mul</td>
<td>f32.mul</td>
<td>f64.mul</td>
</tr>
<tr>
<td>div s</td>
<td>div s</td>
<td>f32.div</td>
<td>f64.div</td>
</tr>
<tr>
<td>div u</td>
<td>div u</td>
<td>f32.abs</td>
<td>f64.abs</td>
</tr>
<tr>
<td>rem s</td>
<td>rem s</td>
<td>f32.remainder</td>
<td>f64.remainder</td>
</tr>
<tr>
<td>rem u</td>
<td>rem u</td>
<td>f32.conversion</td>
<td>f64.conversion</td>
</tr>
<tr>
<td>and</td>
<td>and</td>
<td>f32.call</td>
<td>f64.call</td>
</tr>
<tr>
<td>or</td>
<td>or</td>
<td>f32.file</td>
<td>f64.file</td>
</tr>
<tr>
<td>xor</td>
<td>xor</td>
<td>f32.trunc</td>
<td>f64.trunc</td>
</tr>
<tr>
<td>shr</td>
<td>shr</td>
<td>f32.nearest</td>
<td>f64.nearest</td>
</tr>
<tr>
<td>shr s</td>
<td>shr s</td>
<td>f32.sqrt</td>
<td>f64.sqrt</td>
</tr>
<tr>
<td>rotl</td>
<td>rotl</td>
<td>f32.min</td>
<td>f64.min</td>
</tr>
<tr>
<td>rot r</td>
<td>rot r</td>
<td>f32.max</td>
<td>f64.max</td>
</tr>
<tr>
<td>clz</td>
<td>clz</td>
<td>f32.clz</td>
<td>f64.clz</td>
</tr>
<tr>
<td>ctz</td>
<td>ctz</td>
<td>f32.ctz</td>
<td>f64.ctz</td>
</tr>
<tr>
<td>popcnt</td>
<td>popcnt</td>
<td>f32.popcnt</td>
<td>f64.popcnt</td>
</tr>
<tr>
<td>eqr</td>
<td>eqr</td>
<td>f32.compare</td>
<td>f64.compare</td>
</tr>
<tr>
<td>eq</td>
<td>eq</td>
<td>f32.eq</td>
<td>f64.eq</td>
</tr>
<tr>
<td>ne</td>
<td>ne</td>
<td>f32.ne</td>
<td>f64.ne</td>
</tr>
<tr>
<td>lt s</td>
<td>lt s</td>
<td>f32.lt</td>
<td>f64.lt</td>
</tr>
<tr>
<td>le s</td>
<td>le s</td>
<td>f32.le</td>
<td>f64.le</td>
</tr>
<tr>
<td>lt u</td>
<td>lt u</td>
<td>f32.eq</td>
<td>f64.eq</td>
</tr>
<tr>
<td>le u</td>
<td>le u</td>
<td>f32.ne</td>
<td>f64.ne</td>
</tr>
<tr>
<td>gt s</td>
<td>gt s</td>
<td>f32.gt</td>
<td>f64.gt</td>
</tr>
<tr>
<td>ge s</td>
<td>ge s</td>
<td>f32.ge</td>
<td>f64.ge</td>
</tr>
<tr>
<td>gt u</td>
<td>gt u</td>
<td>f32.gt</td>
<td>f64.gt</td>
</tr>
<tr>
<td>ge u</td>
<td>ge u</td>
<td>f32.ge</td>
<td>f64.ge</td>
</tr>
</tbody>
</table>

#BHUSA @BlackHatEvents
Results: ~6 bugs/vulnerabilities

- Found some new bugs **by accident** during conversion from text format (wat) to binary format (wasm)

- Wasmprinter (Rust)
  - Out of Memory (OOM) error - [issue](#)

  ```
  memory allocation of 4294967296 bytes failed[1] 12038 abort (core dumped)
  ```

- WABT (C++) - wasm2wat, wast2json
  - Assertion failure - [issues](#) (5)

  ```
  SIGABRT:PC.7ffffff8811b.STACK.1b61dc8673.CODE.-6.ADDR.8.INSTR.mov _ 0x108(%rsp),%rax.fuzz
  wast2json: /home/wasm-training/Documents/wasm-tools/wasm2wat/src/wast-parser.cc:1545: waabt: Result waabt::(a
  anonymous namespace)::BinaryWriter::WriteModule()... 0' failed.
  Aborted (core dumped)
  ```
Fuzzing strategy: Improvements #4

- Create **edge case** modules
  - Duplicate sections (unique & customs)
  - Redefinition of exported/imported functions & memory
  - Change sections ordering
  - Create a lot of sections, elements, etc.
  - Inject unusual values for int/float

```
10_memory.wast:6:3: error: only one memory block allowed
    (memory 1 )

10_table.wast:6:3: error: only one table allowed
    (table 1 anyfunc)
```

- Create a **polyglot** WebAssembly module
  - Valid HTML/JS/wasm file
    - Data section injection
    - Custom section injection
  - Detailed blogpost [here](#)

```
unop: ctz | clz | popcnt | ...
binop: add | sub | mul | ...
relop: eq | ne | lt | ...
sign: sju
offset: offset=<nat>
align: align=(1|2|4|8|...)
cvttop: trunc | extend | wrap | ...
val_type: i32 | i64 | f32 | f64
elem_type: funcref
block_type : ( result <val_type>* )* 
func_type: ( type <var> )? <param>* <result>* 
global_type: <val_type> | ( mut <val_type> )
table_type: <nat> <nat>? <elem_type>
memory_type: <nat> <nat>? 
expr:
  ( <op> )
  ( <op> <expr>+ )
  ( block <name>? <block_type> <instr>* )
  ( loop <name>? <block_type> <instr>* )
  ( if <name>? <block_type> ( then <instr>* ) ( if <name>? <block_type> <expr>+ ( then <inst
```
4. Structure-aware fuzzing
Fuzzing strategy: Structure-aware fuzzing

- **Structure-aware fuzzing**
  - Generate semi-well-formed inputs based on knowledge of structure, file format, or protocol.
  - Modules are generated, **without losing time in parsing**, with fuzzy values placed at strategic locations.
Standalone VMs (Rust): Structure-based fuzzing

- **Fuzzers**
  - *Arbitrary trait*: The trait for generating structured data from arbitrary, unstructured input.
  - *wasm-smith*: A WebAssembly test case generator.

- **Targets (all)**
  - Rust code directly **via in-process fuzzing** (cargofuzz, honggfuzz-rs, etc.)
  - Other targets via **shared corpora**

- **Complexity: Low/Medium**
  - Integrating the arbitrary trait can be challenging
  - Wasm-smith is really good, fast and easy to use

- **Results: 0 new direct bugs**
  - Generate interesting inputs that will be mutated later
  - Helps to increase coverage

```rust
#![no_main]

use libfuzzer_sys::fuzz_target;
use wasm_smith::Module;

fuzz_target!(|module: Module| {
    let wasm_bytes = module.to_bytes();

    // Your code here...
});
```
5. Differential fuzzing
Fuzzing strategy: Improvements #5

- **Add new fuzzing harnesses** to target instantiation phases.
  - Create simple imports and provide them to Instance constructors.

```rust
pub fn fuzz_wasmer_instantiate(data: &[u8]) -> bool {
    use wasmer_runtime::{imports, instantiate};
    let import_object = imports! {};
    instantiate(&data, &import_object).is_ok()
}
```

```rust
pub fn wasmi_instantiate(data: &[u8]) -> bool {
    use wasmi::{ImportsBuilder, Module, ModuleInstance};
    match Module::from_buffer(&data) {
        Ok(module) => ModuleInstance::new(&module, &ImportsBuilder::default()).is_ok(),
        _ => false,
    }
}
```

```rust
pub fn fuzz_wasmtime_instantiate_all_cranelift(data: &[u8]) -> bool {
    let store = match get_store_all_feat(Strategy::Cranelift) {
        None => return false,
        Some(a) => a,
    };
    // Create a Module
    let module = match Module::from_binary(&store.engine(), &data) {
        Ok(a) => a,
        _ => return false,
    };
    Instance::new(&store, &module, &[]).is_ok()
}
```
Fuzzing strategy: Differential fuzzing

- **Differential fuzzing**
  - Observe if two program implementations/variants **produce different outputs** for the same input.
  - Really **efficient way to find logic bugs**, unimplemented cases, etc.
  - Famous differential fuzzing projects
    - cryptofuzz, beacon-fuzz

![Diagram showing the comparison of .wasm files for different fuzzing tools](image-url)
Differential fuzzing

- **Type of bugs:**
  - Logic bugs or unimplemented features
  - **Consensus bugs** (critical for blockchains)

- **Fuzzers:** Just a Python or Bash script is working

- **Targets:** All of them

- **Complexity:** **Low**
  - No need for any bindings if you’re using threads/subprocesses
  - A lot of false positives due to WebAssembly feature supports

- **Results:** **2 bugs/vulnerabilities**
  - [wabt] Incorrect validation/rejection - [issues](#)
What about browsers?

![Chrome](image1.png) ![Firefox](image2.png) ![Safari](image3.png)
Targets: Browser’s WebAssembly VMs

- In browsers, the WebAssembly runtime is **part of the JavaScript engine.**

- Targets
  - SpiderMonkey (Firefox)
  - JavaScriptCore (Safari)
  - V8 (Google chrome)
In browsers, the WebAssembly runtime is **part of the JavaScript engine.**

**Targets**
- **SpiderMonkey** (Firefox)
- **JavaScriptCore** (Safari)
- **V8** (Google chrome)
WebAssembly JavaScript APIs

- Complete documentation on Mozilla [MDN for WebAssembly](https://developer.mozilla.org/en-US/docs/WebAssembly)
  - Methods/Constructors
  - Browser compatibility table

---

<table>
<thead>
<tr>
<th>Method</th>
<th>Basic support</th>
<th>CompileError</th>
<th>Global</th>
<th>Instance</th>
<th>LinkError</th>
<th>Memory</th>
<th>Module</th>
<th>RuntimeError</th>
<th>Table</th>
<th>Compile</th>
<th>compile</th>
<th>compileStreaming</th>
<th>instantiate</th>
<th>instantiateStreaming</th>
<th>validate</th>
</tr>
</thead>
<tbody>
<tr>
<td>WebAssembly.instantiate()</td>
<td>57</td>
<td>16</td>
<td>52 *</td>
<td>No</td>
<td>44</td>
<td>11</td>
<td>57</td>
<td>Yes</td>
<td>52 *</td>
<td>?</td>
<td>11</td>
<td>7.0</td>
<td>8.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WebAssembly.instantiateStreaming()</td>
<td>57</td>
<td>16</td>
<td>52 *</td>
<td>No</td>
<td>44</td>
<td>11</td>
<td>57</td>
<td>Yes</td>
<td>52 *</td>
<td>?</td>
<td>11</td>
<td>7.0</td>
<td>8.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WebAssembly.compile()</td>
<td>No</td>
<td>No</td>
<td>62</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
<td>No</td>
</tr>
<tr>
<td>WebAssembly.compileStreaming()</td>
<td>57</td>
<td>16</td>
<td>52 *</td>
<td>No</td>
<td>44</td>
<td>11</td>
<td>57</td>
<td>Yes</td>
<td>52 *</td>
<td>?</td>
<td>11</td>
<td>7.0</td>
<td>8.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WebAssembly.validate()</td>
<td>No</td>
<td>No</td>
<td>62</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
<td>No</td>
</tr>
</tbody>
</table>
WebAssembly JavaScript APIs

- **WebAssembly.Instance**
  - Stateful, executable instance of a WebAssembly.Module.

- **WebAssembly.instantiate**
  - Compile and instantiate WebAssembly code.

- **WebAssembly.instantiateStreaming**
  - Compiles and instantiates a WebAssembly module directly from a streamed underlying source.

- **WebAssembly.Memory**
  - Accessible and mutable from both JavaScript and WebAssembly.

- **WebAssembly.Global**
  - Global variable instance, accessible from both JavaScript and importable/exportable across one or more WebAssembly.Module instances.

- **WebAssembly.Table**
  - Array-like structure accessible & mutable from both JavaScript and WebAssembly.
Fuzzing strategy: Grammar-based fuzzing

- **Grammar-based** fuzzing
  - Javascript files are generated by the fuzzer based on a given grammar
  - We are generating **sequence of WebAssembly JavaScript APIs** calls
  - Fuzzers
    - **Dharma**: Generation-based, context-free grammar fuzzer - wasm.dg
    - **Domato**: DOM fuzzer
    - **Fuzzilli4wasm**: Fuzzer for wasm fuzzing based on fuzzilli

- **Targets**
  - **SpiderMonkey** (Firefox)
  - **JavaScriptCore** (Safari)
  - **V8** (Google chrome)

- **Complexity**: **Medium**
  - You need to manually write grammars
  - It’s time-consuming

- **Results**: **Some bugs & duplicates**
  - Not public
Targets: WebAssembly JIT engines

- **Spidermonkey** (Firefox)
  - WASM-Baseline: fast translation to machine code
  - WASM-Ion: wasm to MIR translator
  - Cranelift: low-level retargetable code generator

- **JavaScriptCore** (Safari)
  - LLInt: Low Level Interpreter
  - BBQ: Build Bytecode Quickly
  - OMG: Optimized Machine-code Generator

- **V8** (Google chrome)
  - Liftoff: baseline compiler for WebAssembly
  - TurboFan: optimizing compiler
Fuzzing strategy: Differential fuzzing

```
.wasm

Liftoff
  Function Body Decoder
  Code Generation

TurboFan
  Function Body Decoder
  Graph Construction (SSA)
  Optimizations
  Scheduling
  Code Generation
  Register Allocation
  Instruction Selection

arg: 42

res: 42
res: 56
```
Fuzzing strategy: Differential fuzzing

- Type of JIT bugs
  - Memory corruption bugs in the compiler
  - Incorrect optimization
  - Bugs in code generators

- Targets
  - WASM-Baseline vs WASM-Ion vs Cranelift
  - LLInt vs BBQ vs OMG
  - Liftoff vs TurboFan

- Complexity: Hard
  - You need to generate valid wasm modules
  - You can force optimization using JS loops

- Results: 0 bugs/vulnerabilities (WIP)
  - JIT compilers for WebAssembly are really simple for the moment
  - Not a lot of public research, it’s still an early stage idea but some non-public bugs have been reported by researchers.
Results & Closing Remarks
Conclusion & Final results

- **Some numbers**
  - ~117 bugs found
    - Rust: 53, C/C++: 53
    - Python: 10, JavaScript: 1
    - Some non-public bugs
  - Final corpora size: ~2M wasm modules
  - Total research time: 2 years
  - Active research time: 6 months full-time
  - ~84 fuzzing harnesses created
  - **WARF**: WebAssembly Runtimes Fuzzing

- **Challenges**
  - Complex to keep everything up-to-date
  - Not the same WebAssembly features are **supported by the VMs**
  - Need to adapt to multiple fuzzing frameworks and languages

- **Future / Next steps**
  - Add new targets and fuzzing harnesses (Go, Java, etc.)
  - **Update fuzzing harnesses** for WebAssembly MVP 2.0
Thanks for your time! Any questions?

- Twitter: @Pat_Ventuzelo
- Mail: patrick@fuzzinglabs.com

#BHUSA @BlackHatEvents