WASM Fuzzing

Zhang Yifan
Department of Computer Science and Engineering
SUStech, Shenzhen, China
11711335@mail.sustech.edu.cn

Yu Tiancheng
Department of Computer Science and Engineering
SUStech, Shenzhen, China
11712019@mail.sustech.edu.cn

Hu Yubin
Department of Computer Science and Engineering
SUStech, Shenzhen, China
11712121@mail.sustech.edu.cn

***Abstract*—WebAssembly, abbreviated as WASM, is a binary instruction format for a stack-based virtual machine. It is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications. It appeared in 2015, and its relatively new, but now has already been extensively used and has been shipped in 4 major browser engines. To nd the bugs in WebAssembly compilers, we made some mutation on WebAssembly on abstract syntax tree level and used some large scale test data to do a fuzzing test on the Web-Assembly virtual machines. We would dynamically execute a program on some test inputs and also the equivalent variants of the original to check whether the outputs are coherent. We also built a WASM Fuzzing Platform for convenient testing and also a better visible demonstration of our research work.**

I. Introduction

A. WebAssembly

1) Introduction: WebAssembly, abbreviated as WASM, is a binary instruction format for a stack-based virtual machine. It is an open standard that defines a portable binary code format for executable programs, and a corresponding textual assembly language, as well as interfaces for facilitating interactions between such programs and their host environment. The main goal of WebAssembly is to enable high-performance applications on web pages, but the format is designed to be executed and integrated into other environments as well. WebAssembly's precursor technologies were asm.js from Mozilla and Google Native Client, and the initial implementation was based on the feature set of asm.js. WebAssembly is used as a complement for JavaScript on the web, and it perfects some functions that JavaScript cannot realize on the web, so it's now shipped in four major browser engines, including Chrome, Firefox and so on. WebAssembly has several advantages; for example, its code size is relatively small, and it has a forced static type that JavaScript does not have. JavaScript has to go through two steps when the browser engine executes code: parser and bytecode compilation, which are the most time-consuming processes. But WebAssembly doesn't have to. It is code that has already been compiled by the compiler, and it was designed aiming to execute at native speed by taking advantage of conventional hardware capabilities available on a wide range of platforms. Also, WASM does not replace JavaScript, to use it in browsers, users may compile C++ source code into a binary file which runs in the same sandbox as regular JavaScript code. So WebAssembly describes a memory-safe, sandboxed execution environment that may be implemented inside existing JavaScript virtual machines. When embedded in the web, WebAssembly will enforce the same-origin and permissions security policies of the browser. In all, it is designed to maintain the versionless, feature-tested, and backward-compatible nature of the web and has a natural advantage when applying on the web.

2) Code representation: There are several representations of WebAssembly. In March 2017, the WebAssembly Community Group reached a consensus on the original binary format, JavaScript API, and reference interpreter. It defines a WebAssembly binary format, which is not designed to be used by humans, and also a human-readable linear assembly bytecode format that resembles traditional assembly languages.

3) Virtual Machine: Wasm code (binary or bytecode) is intended to be run on a portable virtual stack machine, which is the VM. The VM is designed to be faster to parse and execute than JavaScript and to have a compact code representation. A Wasm program is designed to be a separate module containing a collection of various wasm-defined value, and program types definitions expressed either in binary or textual format (see below) that both have a standard structure. The core standard for the binary format of wasm program defines Instruction Set Architecture consisting of a specific binary encoding of types of operations that are executed by the VM. It doesn't specify how exactly they must be executed by the VM, however.

The list of instructions includes standard memory load/store instructions, numeric, parametric, control of flow instruction types, and wasm-specific variable instructions.

B. Abstract Syntex Tree

An abstract syntax tree is a tree model of an entire program or a specific program structure. In our research, AST especially means representing the syntax of WebAssembly code as a hierarchical tree-like structure. The tree focuses on the rules rather than elements like braces or semicolons that terminate statements in some languages. The tree is hierarchical, with the aspects of programming statements broken down into their parts. For example, a tree for a conditional statement has the rules for variables hanging down from the required operator. While transferring the source code to its according to the abstract syntax tree, there are crucial steps like parsing source code and write back.

C. Equivalence Modulo Inputs

EMI is a simple, broadly applicable concept for validating compilers. It aims to take existing real-world code and transform it into different but equivalent variants of the original code. The EMI method is defined as follows: Given a program $P \in L$, any input set $I \subseteq dom(P)$ naturally induces a collection of programs $Q \in L$ that is EMI (w.r.t. I) to P. We call this collection Ps EMI variants. More concretely, given a program P and a set of input values I from its domain, the input set I induces a natural collection of programs C such that every program $Q \in C$ is equivalent to P modulo I: $\forall i \in I$, $Q(i) = P(i)$. The collection C can then be used to perform differential testing of any compiler Comp: If $Comp(P)(i) \neq Comp(Q)(i)$ for some $i \in I$and $Q \in C$, Comp has a miscompilation.

D. Fuzz Testing

Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks. If a vulnerability is found, a software tool called a fuzzer can be used to identify possible causes. It is a quality assurance technique used to discover coding errors and security loopholes in software, operating systems, or networks. Fuzzing works best for finding vulnerabilities that can be exploited by buffer overow, denial of service, cross-site scripting, and SQL injection. It is less useful for dealing with security threats that do not cause program crashes, such as spy-ware, some viruses, worms, Trojans, and key loggers. Although fuzz testing is simple, it offers a high benet-to-cost ratio and can often reveal severe defects that are overlooked when the software is written and debugged. It cannot provide a complete picture of the overall security, quality, or effectiveness of a program, however, and is most effective when used in conjunction with extensive black box testing, beta testing, and other proven debugging methods.

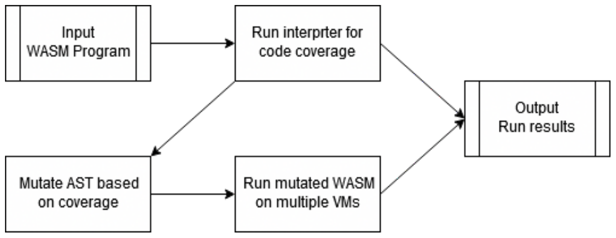


Fig. 1. WASM FUZZING

II. Illustrative Examples

A. Procedure

Our research process consists of mainly three steps.
1. The first step is to convert the WASM code to Abstract Syntax Tree. We complete this by using the official WebAssembly interpreter, and we used the WebAssembly test suite.
2. The second step is to make legit modifications to the ASTs. We would traverse all the nodes and detect the part of code that appears to have no impact on the program and modify or delete them. Then after making sure that other parts of the AST are not affected, we do the final step.
3. The last step is to convert the modified AST back to the WASM code and check the results. We would run this process simultaneously on different WASM virtual machines.

To present our research results, we built a display platform, WASM Fuzzing, which would allow users to upload targeting les. Then the server would access this le. After obtaining, our server would begin modification. It will use our tools to generate equivalent variants of this file and execute them. Then the platform would return the results to the front-end of our platform and display on the website. The code before and after modication will also be displayed, and the platform would highlight the differences. In the last presentation, we were only able to run tests on one WebAssembly virtual machine, which is the spec-interpreter, but now our platform can run the test on three virtual devices, and the other two are wasmer and wastime. Also, in the last presentation, we've only done one WebAssembly code mutation, which is the code coverage check. But now we've managed to add pieces of dead code into the code while guaranteeing that the structure or grammar of the code is not sabotaged.

After mutation, if the output results are different, then there might be something abnormal going on with in the virtual machine. Also, we made some improvements to our WASM fuzzing website. It supports several formats of wast files now because it is more convenient to do this testing. For future research, we would look deeper into the grammar and structure of WebAssembly and make more complex modifications on the WASM ASTs. In the last stage of our research, we anticipate that it might take months to test the massive scale of testing data and try to find bugs within the WebAssembly virtual machines.

```
(∗ Expressions ∗)

type var = int32 Source.phrase
type literal = Values.value Source.phrase
type name = int list

type instr = instr' Source.phrase
and instr' =
  | Unreachable                         (∗ trap unconditionally ∗)
  | Nop                                 (∗ do nothing ∗)
  | Drop                                (∗ forget a value ∗)
  | Select                              (∗ branchless conditional ∗)
  | Block of stack_type ∗ instr list    (∗ execute in sequence ∗)
  | Loop of stack_type ∗ instr list     (∗ loop header ∗)
  | If of stack_type ∗ instr list ∗ instr list  (∗ conditional ∗)
  | Br of var                           (∗ break to n−th surrounding label ∗)
  | BrIf of var                         (∗ conditional break ∗)
  | BrTable of var list ∗ var           (∗ indexed break ∗)
  | Return                              (∗ break from function body ∗)
  | Call of var                         (∗ call function ∗)
  | CallIndirect of var                 (∗ call function through table ∗)
  | LocalGet of var                     (∗ read local variable ∗)
  | LocalSet of var                     (∗ write local variable ∗)
  | LocalTee of var                     (∗ write local variable and keep value ∗)
  | GlobalGet of var                    (∗ read global variable ∗)
  | GlobalSet of var                    (∗ write global variable ∗)
  | Load of loadop                      (∗ read memory at address ∗)
  | Store of storeop                    (∗ write memory at address ∗)
  | MemorySize                          (∗ size of linear memory ∗)
  | MemoryGrow                          (∗ grow linear memory ∗)
  | Const of literal                    (∗ constant ∗)
  | Test of testop                      (∗ numeric test ∗)
  | Compare of relop                    (∗ numeric comparison ∗)
  | Unary of unop                       (∗ unary numeric operator ∗)
  | Binary of binop                     (∗ binary numeric operator ∗)
  | Convert of cvtop                    (∗ conversion ∗)
```

Fig. 2. WebAssembly Expressions

III. EMI and Platform Implementation

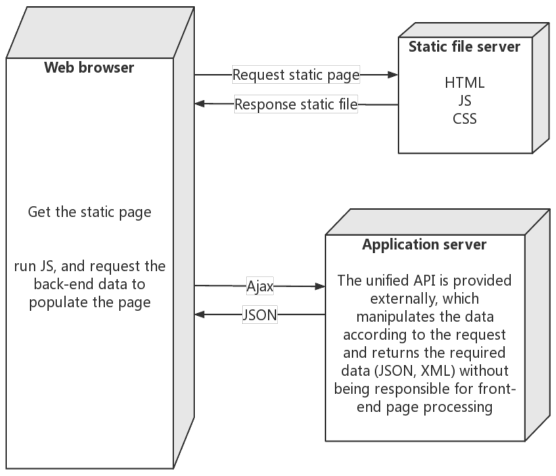
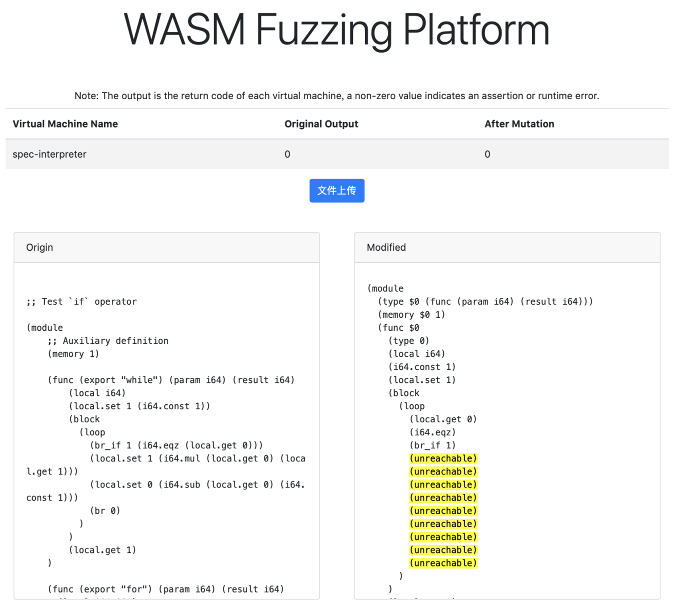

Web browser
Get the static page
run JS, and request the back-end data to populate the page
Request static page
Response static file
Static file server
HTML
JS
CSS
Ajax
JSON
Application server
The unified API is provided externally, which manipulates the data according to the request and returns the required data (JSON, XML) without being responsible for front-end page processing

Fig. 3. Platform Structure

A. Introduction

We built our test platform by separating front and back ends. Separation of front and back ends has become the industry standard use method for Internet project development. It can be decoupled effectively by nginx + tomcat (and can also add a nodejs in between) this way, the front end and the back end have lower coupling. The core idea is that the front-end HTML page calls the back-end rest API interface via ajax and interacts with JSON data. The advantages of this approach are


WASM Fuzzing Platform

Note: The output is the return code of each virtual machine, a non-zero value indicates an assertion or runtime error.

Virtual Machine Name	Original Output	After Mutation
spec-interpreter	0	0

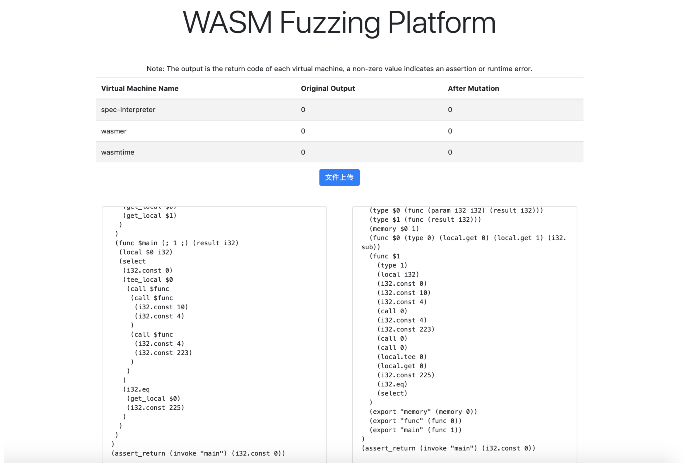
Origin

```
;; Test `if` operator

(module
    ;; Auxiliary definition
    (memory 1)

    (func (export "while") (param i64) (result i64)
        (local i64)
        (local.set 1 (i64.const 1))
        (block
          (loop
            (br_if 1 (i64.eqz (local.get 0)))
            (local.set 1 (i64.mul (local.get 0) (loca
l.get 1)))
            (local.set 0 (i64.sub (local.get 0) (i64.
const 1)))
            (br 0)
          )
        )
        (local.get 1)
    )

    (func (export "for") (param i64) (result i64)
```

Modified

```
(module
  (type $0 (func (param i64) (result i64)))
  (memory $0 1)
  (func $0
    (type 0)
    (local i64)
    (i64.const 1)
    (local.set 1)
    (block
      (loop
        (local.get 0)
        (i64.eqz)
        (br_if 1)
        (unreachable)
        (unreachable)
        (unreachable)
        (unreachable)
        (unreachable)
        (unreachable)
        (unreachable)
        (unreachable)
        (unreachable)
      )
    )
```

Fig. 4. WASM Fuzzing Platform Version 1


WASM Fuzzing Platform

Note: The output is the return code of each virtual machine, a non-zero value indicates an assertion or runtime error.

Virtual Machine Name	Original Output	After Mutation
spec-interpreter	0	0
wasmer	0	0
wasmtime	0	0

```
    (get_local $1)
   )
  )
  (func $main (; 1 ;) (result i32)
   (local $0 i32)
   (select
    (i32.const 0)
    (tee_local $0
     (call $func
      (call $func
       (i32.const 10)
       (i32.const 4)
      )
      (call $func
       (i32.const 4)
       (i32.const 223)
      )
     )
    )
    (i32.eq
     (get_local $0)
     (i32.const 225)
    )
   )
  )
 )
 (assert_return (invoke "main") (i32.const 0))
```

```
  (type $0 (func (param i32 i32) (result i32)))
  (type $1 (func (result i32)))
  (memory $0 1)
  (func $0 (type 0) (local.get 0) (local.get 1) (i32.
sub))
  (func $1
    (type 1)
    (local i32)
    (i32.const 0)
    (i32.const 10)
    (i32.const 4)
    (call 0)
    (i32.const 4)
    (i32.const 223)
    (call 0)
    (call 0)
    (local.tee 0)
    (local.get 0)
    (i32.const 225)
    (i32.eq)
    (select)
  )
  (export "memory" (memory 0))
  (export "func" (func 0))
  (export "main" (func 1))
)
(assert_return (invoke "main") (i32.const 0))
```

Fig. 5. WASM Fuzzing Platform Version 2

apparent. The front end does not embed any background code. The front end focuses on the development of HTML, CSS, and js, and does not depend on the back end. We can also simulate JSON data to render the page. When we find bugs, we can quickly locate who is the problem, and there will be no mutual detachment. And it will help us continue to develop on this basis.

B. Use of Platform

We built a WASM Fuzzing Platform and deployed it on our server. You can visit the website to upload a file on your own: http://wasm.eveneko.com By visiting the above website and use our WASM Fuzz Platform, we can upload any file of the ”wast” format, and our server would save and modify this file. Then the platform will display the output result of the original archive as well as the modified file. The ouput is the return code of each WASM virtual machine, so 0 means the two results conforms. Any non-zero output indicates there exits an assertion or runtime error. Also, the code before and after modification will be displayed, and we would highlight the differences, so it would be easier to observe.

C. Highlights of the platform

1) Support multi-virtual machine operation: Now our platform supports operation on three WebAssembly virtual machines: spec interpreter, wastime, wasmer. When files uploaded, the original files and mutants are simultaneously executed on all these three WebAssembly virtual machines and all results are displayed. By this way, we can do this test on more VMs and get more feedback, and increasing the possibility of revealing bugs.

2) Binary format uploading: We spotted some errors when uploading our test files. We found the reason was that there's lack of uniformity of the format of wast file. So we refined our platform and changed the uploading format to binary form. Then the files can be operated regardless of instruction formats.

3) Highlighting difference: When file uploading and execution are successful, the results will be displayed at the platform. Also, the content before and after modification will also be shown, and the differences will be highlighted. This provides great convenience for us to analyze the content and detect the latent problems.

4) Multi-modification supporting: Now we can do two modification on the WebAssembly abstract syntax trees. First is to check the code coverage during execution. Second is to add pieces of dead code to the files while maintaining its structure and grammar. Both are equivalent modulo inputs and the execution results should be consistent with the original file.

IV. Future Work

Now our platform can run the test on three virtual machines; apart from the spec-interpreter, we can also run tests on wasmer and wastime simultaneously. Also, we can do code coverage check, as well as add pieces of dead code into the code while guaranteeing that the structure or grammar of the code is not sabotaged.

For future research, we would look deeper into the grammar and structure of WebAssembly and make more valid modications to the WebAssembly Abstract Syntax Trees. In the last stage of our research, we anticipate that it might take months to test the large scale of testing data and try to find bugs within the WebAssembly virtual machines.

References

[1] Vu Le, Mehrdad Afshari, and Zhendong Su, "Compiler Validation via Equivalence Modulo Inputs," April 2014.
[2] Nlsandler, https://github.com/nlsandler/write_a_c_compiler
[3] WebAseembly, spec. https://github.com/WebAssembly/spec