electronics

MDPI

Article

ESFuzzer: An Efficient Way to Fuzz WebAssembly Interpreter

Jideng Han [1,*], Zhaoxin Zhang [1], Yuejin Du [2], Wei Wang [1] and Xiuyuan Chen [3]

[1] Faculty of Computing, Harbin Institute of Technology, Harbin 150006, China; heart@hit.edu.cn (Z.Z.); wangwei_hit@hit.edu.cn (W.W.)
[2] Beijing Qihoo Technology Co., Ltd., Beijing 100015, China; du.yuejin@gmail.com
[3] Chengdu Junzheng Technology Co., Ltd., Chengdu 610031, China; fqhshmily@163.com
* Correspondence: 18b903075@stu.hit.edu.cn

Abstract: WebAssembly code is designed to run in a sandboxed environment, such as a web browser, providing a high level of security and isolation from the underlying operating system and hardware. This enables the execution of untrusted code in a web browser without compromising the security and integrity of the user's system. This paper discusses the challenges associated with using fuzzing tools to identify vulnerabilities or bugs in WebAssembly interpreters. Our approach, known as ESFuzzer, introduces an efficient method for fuzzing WebAssembly interpreters using an Equivalent-Statement concept and the Stack Repair Algorithm. The samples generated by our approach successfully passed code validation. In addition, we developed effective mutation strategies to enhance the efficacy of our approach. ESFuzzer has demonstrated its ability to generate code that achieves 100% WebAssembly validation testing and achieves code coverage that is more than twice that of libFuzzer. Furthermore, the 24-hour experiment results show that ESFuzzer performs ten times more efficiently than libFuzzer.

Keywords: fuzzing; WebAssembly interpreter; equivalent-statement; equivalent-exchange

check for updates

Citation: Han, J.; Zhang, Z.; Du, Y. Wang, W.; Chen, X. ESFuzzer: An Efficient Way to Fuzz WebAssembly Interpreter. *Electronics* **2024**, *13*, 1498. https://doi.org/10.3390/electronics13081498

Academic Editor: George Angelos Papadopoulos

Received: 2 March 2024
Revised: 5 April 2024
Accepted: 8 April 2024
Published: 15 April 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

WebAssembly is a binary instruction format designed for a virtual machine that operates within web browsers. It serves as a portable target for compiling high-level programming languages, such as C/C++, Rust, Java, and others. Its interpreter is a low-level, stack-based machine. The primary goal of WebAssembly is to offer a secure, efficient, and platform-independent approach to executing code in web browsers (97.14% of web browsers supported it in April 2024, see https://caniuse.com/wasm, accessed on 11 April 2024), allowing web applications to handle computationally intensive tasks. WebAssembly is widely supported in modern web browsers, including Chrome, Firefox, Safari, and Edge. Its usage is becoming increasingly prevalent in various applications. However, this exposes new attack surfaces in the browser, and errors in the WebAssembly interpreter may result in incorrect program outputs or abnormal termination, potentially leading to security vulnerabilities. In recent years, several security issues have been identified, including vulnerabilities discovered by Natalie Silvanovich in Webkit [1] and seven CVEs reported for WAVM in 2018 [2].

Fuzzing has proven to be effective at identifying vulnerabilities [3], and its application extends to detecting vulnerabilities in WebAssembly interpreters, similar to its use in other high-level programming languages like JavaScript and C/C++. However, disparities emerge when comparing WebAssembly with these languages, particularly in the area of validation. In WebAssembly, the validation process carefully evaluates each instruction within the program input to ensure the integrity of the type and quantity of the values present on the stack. This particular characteristic poses challenges for traditional fuzzing methodologies, such as AFL [4] and libFuzzer [5], as generating valid testcases becomes a more complex task. Consequently, the testing process becomes less efficient and effective,

necessitating the exploration of alternative approaches to address this limitation within the context of academic research.

To address this demand, we propose a fuzzing framework named ESFuzzer. The contributions of our approach are as follows:

- We introduce the concepts of Equivalent-Statement and Equivalent-Exchange, along with a novel intermediate representation to present them. These concepts provide valuable insights into program behavior, allowing us to optimize, transform, and analyze programs more effectively.
- We have developed a novel algorithm called the Stack Repair Algorithm. This algorithm tackles challenges associated with stack manipulation in the fuzzing process.
- We have developed a fuzzing framework called ESFuzzer and conducted an evaluation using libFuzzer on Chrome for V8. The results demonstrate that our tool effectively addresses the validation challenge and is more efficient than libFuzzer.

2. Background and Related Works

When conducting fuzz testing for WebAssembly interpreters, researchers encounter several challenges. Some of these challenges arise from the fuzz testing technique itself. Others include the execution environment and the semantic complexity of WebAssembly as a low-level bytecode format, which introduce additional challenges to the testing process. To find related works on this topic, a search was conducted using the keywords "WebAssembly interpreter" and "fuzzing", with the operator "AND".

2.1. Fuzzing Overview

Fuzzing is a popular area of research that has received a lot of attention in recent years and has proven to be effective. Previous research has applied fuzzing techniques to interpreters and compilers using various approaches, such as fuzzing special-purpose languages like C [6–8] or JavaScript [9–12], as well as fuzzing general-purpose languages. The choice of approach depends on the requirements of the target system and the availability of input models. Additionally, language-agnostic tools have been developed, which do not make any assumptions about the target language and are more widely applicable [12–15]. These findings demonstrate the versatility of fuzzing techniques in identifying vulnerabilities in a wide range of software systems.

A fuzz test begins by generating program inputs, known as testcases. The quality of these testcases directly affects the effectiveness of the test. The inputs should meet the program's requirements for the input format, while also being varied enough to potentially cause the program to fail. The target programs accept various inputs, such as files with different formats, network communication data, and executable binaries with specified characteristics.

How to generate broken-enough testcases is the main challenge for fuzzers. Van Sprundel [16] classified these fuzzers into two types: Generative-based and Mutation-based.

a Generative-based fuzzing: Generative-based approaches use input models such as configuration files [6,13,14,17–21] or static analysis [22,23] to generate valid examples. The main advantage of the generative approach is that the produced inputs are syntactically correct by design, since the generator functions respect the underlying syntax expected by the program under test.

b Mutation-based fuzzing: Mutation-based approaches require a valid set of initial inputs and generate testcases by mutating these inputs. Examples of mutation-based fuzzers include Honggfuzz [24], AFL++ [25], Collafl [26], POSTER [27], AFLFast [28], and Angora [29], which collect metadata about the execution of a target program using compile-time instrumentation. This classification helps us understand the different approaches to fuzzing and their suitability for different types of software systems.

Particularly in the field of fuzz testing for interpreters or compilers, prior research has focused on generating input samples for specific ones. Lindig's [8] consistency checker for the C language compiler generates C code using a small grammar and fixed test generation

scheme. Yang et al. introduced CSmith [6], a language-specific fuzzer for testing compilers. The paper's primary contributions are twofold: First, it advances the state of the art in compiler testing by creating programs that cover a large subset of C while avoiding undefined and unspecified behaviors that could hinder the automatic detection of wrong-code bugs. Second, it presents a collection of qualitative and quantitative results regarding the bugs found in open-source C compilers. Ruderman's jsfunfuzz [30] is a generation-based fuzzer trained to handle various JavaScript language features. Domato [31] generated samples targeting specific DOM logic issues by using HTML, CSS, and JavaScript syntax. CodeAlchemist [32] employed a semantics-aware assembly approach to produce JavaScript code snippets. Additionally, some works [10,15,33] have focused on the abstract syntax tree (AST) of JavaScript, such as Park's work [12], which introduced the concept of aspect-preserving mutations. These techniques demonstrate the diverse approaches to fuzzing compilers and interpreters, highlighting the importance of developing specialized fuzzers for specific languages and systems.

Another challenge is low coverage. The aim of fuzz testing is to cover as many execution paths and code regions of the target system as possible to reveal hidden vulnerabilities. However, real software systems are often complex, with numerous execution paths and code branches. Several studies have explored this challenge. Skyfire [34] uses data-driven seed generation, while Dharma [35] and F1 [36] optimize the input generation process from the grammar itself. Vuzzer [37] extracts data-flow and control-flow features to create a smart feedback loop, while other studies combine coverage feedback with syntax awareness. Nautilus [18] combines coverage feedback with syntax awareness and mutation operators inspired by AFL, and Zest [32] uses a Quickcheck-like [38] input generator for coverage-guided fuzzing. Montage [39] leverages machine learning to find bugs by combining AST fragments in unique ways.

2.2. WebAssembly Fuzzing

WebAssembly is a new technique that is currently under research with a focus on security. Some articles [40–46] have looked into the security of WebAssembly binaries, such as Daniel Lehmann's Fuzzm [42], the first binary-only grey-box fuzzer for WebAssembly. This approach uses canary-based binary instrumentation to detect overflows and underflows on the stack and heap. Other approaches [47–49], such as that by Dwfault [47], have introduced a mechanism of hierarchical variation and instruction correction based on AFL, which partly solves the problem of syntax validation but introduces a large number of nop instructions that can break the program's logic. Researches [50,51] focus on WebAssembly interpreters in smart contracts. Paper [50] designed and implemented WASAI, a new concolic fuzzer for uncovering vulnerabilities in WebAssembly smart contracts. Paper [51] implemented a grey-box fuzzer called GFuzzer based on WebAssembly for smart contracts on the EOSIO platform considering that EOSIO contracts are not open-sourced.

In addition to smart contracts, WebAssembly is also widely used in web browsers like Firefox, Safari, and Chrome. These browsers come pre-installed with a WebAssembly interpreter. However, there is a lack of research on fuzz testing the WebAssembly interpreter in browsers.

2.3. Understanding WebAssembly

WebAssembly is a binary instruction format specifically designed for a stack-based virtual machine. This enables the execution of code written in multiple languages on the web at nearly native speed, making it possible for client applications that were previously unable to be run on the web. WebAssembly programs are usually written in high-level programming languages like C/C++ and Rust, or in text format, and then compiled into a binary format for distribution on a web server.

WebAssembly has two concrete representations: One is a compact binary format that uses the extension ".wasm". ".wasm" is the typical distribution format for WebAssem-

bly code, and it has a human-readable text format with the extension ".wat" (short for "WebAssembly Text Format").

In both binary and textual formats, the fundamental unit of code in WebAssembly is a module. In textual format, a module is represented as a large S-expression, as shown in Listing 1. A module contains definitions of types, functions, tables, memories, and globals.

Listing 1. A WebAssembly program in text format.

```
(module
    (func (export "addTwo") (param i32 i32) (result i32)
        local.get 0
        local.get 1
        block
            ...
            block
                ...
            end
            ...
        end
        i32.add
    )
)
```

Before a WebAssembly program execution, it is validated by an interpreter. Validation of the WebAssembly code is crucial in the execution process, as it ensures that only valid programs can be instantiated. In contrast to traditional programming languages, WebAssembly's validation procedure extends beyond syntax analysis. This is due to its stack-based virtual machine design, which necessitates the validation of both syntax and the validity of the operands stored on the stack. As a result, the WebAssembly interpreter ensures not only the syntactic correctness of the code, but also its semantic integrity and security.

The validation algorithm of WebAssembly uses two distinct stacks: a value stack and a control stack. The value stack keeps track of the types of operand values on the stack, while the control stack manages structured control instructions and their associated blocks.

When a value is pushed, its type is recorded on the value stack. If the type is unknown, it is marked as "Unknown". When a value is popped, the stack checks that it does not cause underflow in the current block, and then removes one item. This ensures that the program is semantically correct and that the values are of the expected type.

Upon successful validation, a WebAssembly program proceeds to execution within the WebAssembly interpreter. The execution process involves several essential steps:

i Virtual Machine (VM) Initialization: the interpreter initializes the WebAssembly VM and sets up the stack, which serves as a storage mechanism for instruction operands.
ii Instruction Parsing: the WebAssembly VM reads and parses instructions from the program, extracting opcodes, operands, and relevant information.
iii Instruction Execution: the VM executes the corresponding operation based on the opcode and operand, performing the required computation.
iv State Management: after executing an instruction, the VM updates various components, such as program counters, stacks, and other relevant state information, to ensure the continuity of the execution process.
v Control Flow Handling: The interpreter manages the program's control flow by handling branching, looping, and function calls. It keeps track of program counters and manages conditional and unconditional jumps to different program sections.

3. Conception

Before delving into our proposed methodology, it is imperative to establish a conceptual foundation by formally defining several key notions that underpin the methodologies presented in the subsequent sections.

Definition 1. *The Effect-Array use to represent the effect of an instruction on the stack. We can use a two-dimensional array denoted as E. This array encompasses the impact of instructions on the stack, considering both outgoing and incoming operands. Specifically, we define "O" as the outgoing array and "I" as the incoming array. These arrays represent the types and orders of operands in an instruction. The term "o_n" denotes the type of the nth outgoing operand, while "i_n" denotes the type of the nth incoming operand. Currently, there are only four available number types in WebAssembly: i32, f32, i64, and f64.*

$$E = [O, I], \quad O = [o_1, o_2 \dots o_n], \quad I = [i_1, i_2 \dots i_n]$$
$$o_n, i_n \in \{i32, f32, i64, f64\}$$

By utilizing this two-dimensional array E, we can capture and analyze the stack behavior that results from the execution of instructions. This representation enables us to compare and assess the impact of different instructions on the stack. It helps identify appropriate replacements and ensures the preservation of stack integrity throughout the program's execution.

For instance, as shown in Figure 1, the instruction *i*32.*add* removes two operands of type i32 from the stack, computes their sum, and then places the resulting value (of type i32) back onto the stack. Therefore, the Effect-Array for *i*32.*add* can be represented as $E(i32.add) = [[i32, i32], [i32]]$. This indicates that the instruction requires two i32 operands as inputs and generates a single i32 value as the output.

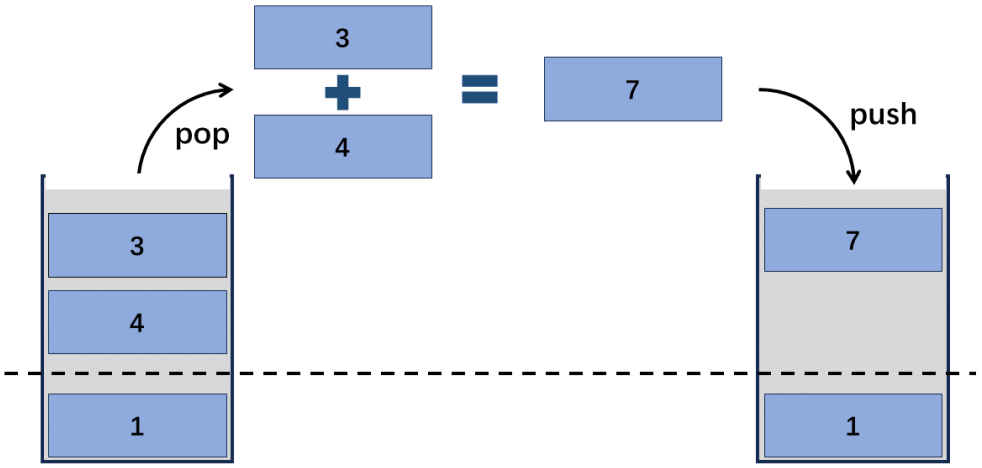


Figure 1. Schematic diagram of the stack operation of *i*32.*add*.

For a given code block, such as Listing 2, the signature of the code block indicates that it requires two i32 parameters and returns an i32 result. Based on this information, we determine that the Effect-Array is $E(block) = [[i32, i32], [i32]]$.

Listing 2. WebAssembly code block.

```
(block (param i32 i32) (result i32)
    local.get 0
    local.get 1
    i32.add
)
```

Definition 2. *An Equivalent-Statement is a set of instructions or code blocks that produce the same Effect-Array.*

An instruction or code block can be considered equivalent if it has the same characteristics in terms of the type, number, and order of its input and output operands on the stack. This equivalence ensures that replacing one statement with another from the set of Equivalent-Statements maintains the stack behavior and overall functionality of the program.

As previously noted, the i32.add instruction and the code block in Listing 2 produce identical Effect-Arrays $[[i32, i32], [i32]]$, demonstrating their equivalence. So, we can say that i32.add and the code block in Listing 2 belong to the same Equivalent-Statement.

Based on the impact of instruction and the code block on the stack, we can classify Equivalent-Statement into three categories:

- Expanded Statement: statements have more out-stack operands than in-stack operands ($len(O) > len(I)$), which causes an increase in the stack when inserted into a program.
- Common Statement: statements have an equal number of out-stack and in-stack operands ($len(O) = len(I)$), resulting in no change to the stack.
- Contracted Statement: statements have fewer out-stack operands than in-stack operands ($len(O) < len(I)$), resulting in a reduction in the stack when inserted into a program.

In program structure, we often observe the following pattern: At the outset, there are numerous Expanded Statements that push a large number of operands onto the stack. Common Statements are interspersed throughout the program to represent routine computations. Towards the end, there are Contracted Statements that consume the stack, leaving only the correct number and type of operands.

By categorizing Equivalent-Statements into these groups, we can easily manage their effects on the stack and streamline the generation of Equivalent-Statements in a program.

For the sake of simplicity, we developed a compact intermediate representation (IR) to depict Equivalent-Statements. An IR instruction looks like this:

$$IR.OP \quad O : [o_1, o_2 \dots o_n], I : [i_1, i_2 \dots i_n]$$

For example, an intermediate instruction with the format *IR.OP O:[i32, i32], I:[i32]* denotes an Equivalent-Statement that pops out two operands of type i32 from the stack and pushes one operand of type i32 back, such as $i32.add$ or $i32.sub$.

Definition 3. *The Equivalent-Exchange involves replacing an instruction or code block with any statement from its corresponding Equivalent-Statement, while maintaining the stack configuration of the program.*

This replacement does not alter the stack behavior or the overall functionality of the program, and it provides valuable flexibility in code transformations and optimizations. It also enables the generation of diverse testcases while maintaining the desired stack behavior. This technique plays a crucial role in generating and mutating testcases for fuzzing.

4. Stack Repair Algorithm

As mentioned earlier, the WebAssembly interpreter conducts the validation process before the program execution. This verification ensures that the type and quantity of values on the top of the stack are congruent with the corresponding instructions. Only samples that pass this validation check are allowed to proceed to the execution phase, while those that fail are rejected. In response to this, we developed a methodology known as the Stack Repair Algorithm, with the goal of improving the success rate of the validation check for the provided samples.

The Stack Repair Algorithm is used to restore the correct stack state after generation or mutation. This was implemented to ensure that the sample passed validation by the

WebAssembly interpreter. This was accomplished by calculating the Effect-Array of each Equivalent-Statement in the sample.

Algorithm 1 presents a systematic approach to stack repair that involves maintaining a virtual stack. At the beginning, the algorithm initializes the virtual stack to match its initial state. Subsequently, it iterates through each statement in the queue, individually evaluating the correctness of the current stack state in relation to the statement's Effect-Array.

To verify the correctness of the stack state, the algorithm compares the virtual stack with the expected Effect-Array associated with the statement. If the virtual stack aligns with the expected Effect-Array, the algorithm proceeds to the next step. If the virtual stack does not match the expected Effect-Array, the algorithm initiates a repair process to indicate an incorrect stack state.

Algorithm 1 Stack Repair

Input: ops
Output: ops

```
1: virtual_stack = initStack()
2: for op in ops do
3:     if staticCheck(virtual_stack, op) then
4:         continue
5:     else
6:         repair(ops)
7:     end if
8: end for
```

During the stack repair process, two main situations may require fixing:

- If the number of operands on the stack is less than what the next statement requires, the algorithm can insert the necessary statement to push the required number of operands onto the stack.
- If the type or order of the operands on the stack is incorrect or does not match what the next statement expects, the algorithm needs to perform a stack rollback. This involves locating and modifying the statement that pushed the incorrect operand onto the stack. Once the correct operand is pushed, the stack can be rolled forward to resume normal execution.

The first case is simpler and does not require an example, whereas the second case is more specific. Therefore, an instance will be provided.

Listing 3 contains a code error on line 4. The WebAssembly interpreter validation will detect an error because for IR.OP O:[[i64,i64], I:[i64]] requires two i64 operands, but the current stack only contains i32 and i64 operands. To fix this issue, stack rollback is necessary, which can be achieved by reversing the execution order of statements and the outgoing and incoming operands.

Starting from line 4, the algorithm first clears the temporary stack and then adds the two i64 operands to it. On line 3, an i64 is taken from the stack and placed into the stack as an i32. The stack now holds [i64, i32]. Line 2 removes an i32 from the stack, leaving only [i64]. The error occurred due to a type mismatch. We remove i64 and replace it with i32 in the stack. The operation should be reversed, resulting in IR. OP O:[i32], I :[i64]. The final result is displayed in Listing 4.

Listing 3. IR code fragment before repair.

```
1 IR.OP O:[] , I[i32]          ;;stack[i32]
2 IR.OP O:[] , I[i32]          ;;stack[i32, i32]
3 IR.OP O:[i32], I[i64]        ;;stack[i32, i64]
4 IR.OP O:[i64, i64], I[i64]   ;;error!
```

Listing 4. IR code fragment after repair.

```
IR.OP O:[] , I[i32]          ;;stack[i32]
IR.OP O:[i32], I[i64]        ;;stack[i64] insert here!
IR.OP O:[] , I[i32]          ;;stack[i64, i32]
IR.OP O:[i32], I[i64]        ;;stack[i64, i64]
IR.OP O:[i64, i64], I[i64]   ;;stack[i64]
```

5. ESFuzzer

After discussing the main concepts and algorithm proposed in this paper, we will now shift our focus to our tool, ESFuzzer. In this section, we will provide a detailed overview of the architecture and implementation of ESFuzzer.

5.1. Overview

ESFuzzer 's workflow is illustrated in Figure 2 and comprises three phases, Generation, Mutation, and Execution. In the Generation phase, ESFuzzer randomly combines intermediate instructions from Corpus to create an intermediate program and send it to the next phase. During the Mutation phase, ESFuzzer mutates the intermediate programs based on the mutation strategy we designed and the process information obtained from the Execution phase; subsequently, ESFuzzer repairs the results using the Stack Repair Algorithm and translates them into a WAT format script for compilation and execution. Finally, in the Execution phase, ESFuzzer monitors the execution of the WebAssembly interpreter and collects test samples that can trigger a crash or explore new paths.

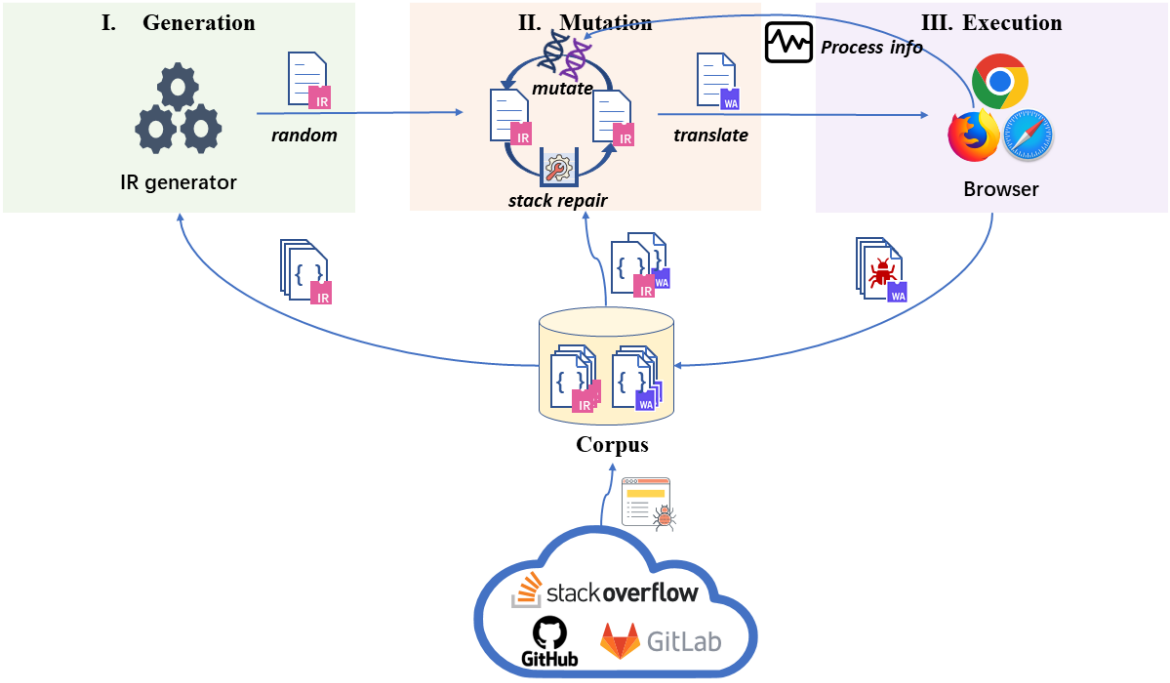


Figure 2. Main workflow of ESFuzzer.

Listing 5 presents a real-world test case generated by ESFuzzer based on Listing 4. It is executable by the WebAssembly interpreter.

Listing 5. A real-world test case generated by ESFuzzer.

```
(module
    (func (param i32 i32) (param i64)
        i32.const 18103
        i64.extend_i32_s
        if (result i32)  ;; label = @1
            i32.const 20160
        else
            i32.const 7257
        end
        i64.extend_i32_u
        i64.sub
    )
    (export "main" (func 0))
)
```

5.2. Corpus

Furthermore, we establish a corpus to store the associations between IR instructions and WebAssembly statements. Each IR instruction is associated with a set of WebAssembly instructions or code blocks. This corpus facilitates efficient translation and ensures the accurate representation of IR instructions in WebAssembly during the Generation and Mutation processes.

ESFuzzer uses the corpus to store code snippets as input. The corpus is sourced from two places: the internet, where code snippets are obtained from public repositories, forums, blogs, and documentation, and refined samples that trigger new paths or crashes.

5.3. Generation

During the *Generation* phase, the tool randomly combines intermediate instructions to create a program. This process involves selecting instructions from a pool of available options and arranging them in a sequence. The resulting program may contain inconsistencies or errors in the stack behavior.

To tackle these issues, we implement our Stack Repair Algorithm. The algorithm systematically analyzes the generated program, checking the correctness of the stack state at each statement based on the expected Effect-Array. If any inconsistencies or errors are detected, the algorithm performs repairs to align the stack state with the expected behavior.

5.4. Mutation

During the Mutation phase, the tool uses three important mutation methods to modify the IR program, based on program information received from the executor. These methods play a crucial role in the program's transformation process.

5.4.1. Statement-Based Mutation

The statement-based mutation offers three options for modifying programs:

- The first option involves exchanging the WebAssembly instructions or code blocks that correspond to the same IR instructions. This allows for the replacement of specific sections of the program while maintaining the overall structure and behavior.
- The second option is to randomly mutate the operands within the IR instructions. This involves modifying the values or variables used in the IR instructions (e.g., $IR.OP[[], [1]] -> IR.OP[[], [978]]$), introducing variations in the program's data flow and computation.
- The third option combines both the exchange of WebAssembly instructions or code blocks and the random mutation of operands within the IR instructions. This compre-

hensive approach allows for a wider range of modifications to the program, increasing the potential for creating new and diverse program instances.

By offering these three options, statement-based mutation provides flexibility for modifying programs, facilitating the exploration of different program configurations and behaviors.

5.4.2. Size-Based Mutation

To effectively manage program complexity, we employ three categories of Equivalent-Statements as discussed in Section 3: Expanded Statement, Common Statement, and Contracted Statement.

To increase the complexity of a program, we can add more Expanded Statements at the beginning or introduce Common Statements in the middle. Conversely, reducing the number of Expanded Statements decreases program complexity. These strategies enable the fine-tuning of the complexity of the program's computation and control flow.

5.4.3. Control Flow-Based Mutation

In the representation of a WebAssembly module, it is typically expressed as a single S-expression (https://en.wikipedia.org/wiki/S-expression, accessed on 11 April 2024). Within this S-expression, functions are defined, and each function contains a sequential list of instructions in its body. This structure enables the modification of the program's control flow by altering the sequence of IR instructions.

By rearranging the order of IR instructions within the linear list of instructions in a function's body, we can effectively modify the program's control flow. This flexibility enables us to rearrange the execution order of instructions and introduce conditional branching, or implement loops, among other control flow modifications.

This trade-off, in which the instruction sequence of the IR is modified to reflect changes in the program's control flow, allows for dynamic and flexible program execution. By manipulating the instruction sequence in the IR, we can customize the program's behavior and achieve the desired control flow patterns within the WebAssembly module.

6. Evaluation

In this section, we will assess our approach by addressing three questions:

- **RQ1:** Did our tool effectively meet the validation challenge?
- **RQ2:** Has the code coverage improved with our tool?
- **RQ3:** Has the efficiency of fuzzing improved with the use of our tool?

6.1. Experimental Setup

6.1.1. Contrast

Coverage-guided fuzzing (CGF) is a powerful testing technique for identifying a wide range of bugs in software applications. One of the most well-known tools based on CGF is libFuzzer, an in-process, coverage-guided, evolutionary fuzzing engine. Due to its in-process nature, libFuzzer enables rapid testing speeds, and its coverage-guided approach makes the testing process highly efficient. As a result, libFuzzer is a powerful tool that has helped uncover thousands of bugs in real-world programs. To ensure an accurate evaluation of our approach, we chose libFuzzer as our experimental control.

6.1.2. Repetitions and System Configuration

To address the variability in results stemming from the inherent non-determinism of fuzzing, we conducted three rounds of fuzzing on the WebAssembly interpreter in V8 using both ESFuzzer and libFuzzer [5]. The experiments were conducted using two machines, each equipped with an Intel Core i7-6700 four-core eight-thread CPU running at 3.4 GHz, 64 GB of RAM, and Ubuntu 20.04 LTS. For libFuzzer, we used LLVM version 13.0.0, and for V8, the version was 9.2.88.

Additionally, to maintain consistency in our experimental design, we limited the use of libFuzzer to the WebAssembly module of V8. Using it directly on V8 would introduce mutations that are inconsistent with our research goals. Fuzzing the WebAssembly module with libFuzzer allowed us to take full advantage of its in-process and coverage-guided features. This ensured that any discrepancies between the results of ESFuzzer and libFuzzer were attributable to variations in the fuzzing techniques rather than the execution environment. To prepare for our experiments, we made some modifications to the V8 codebase:

- We replaced the default LLVM compiler used by V8 with a complete version to ensure the proper functionality of libFuzzer. We disabled the Chrome Clang plugin by setting the "clang_use_chrome_plugins" flag to false, as it could potentially cause compatibility issues with the tool.
- We modified the entry process to work with libFuzzer, which is designed as a library. To accomplish this, we removed the link to the V8/test/fuzzer/fuzzer.cc file and added the "fsanitize=fuzzer" compiler option in the ninja build file "obj/V8_simple_wasm_fuzzer.ninja". This enabled libFuzzer to exclusively work with the WebAssembly V8 module without interfering with other parts of the V8 codebase.

6.2. Results and Analysis

RQ1: Did our tool effectively address the validation challenge?

The "WebAssembly.validate()" (https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface/validate, accessed on 11 April 2024) function is commonly used in WebAssembly testing and evaluation to verify the successful validation of code for testcases. This function accepts a WebAssembly binary as input and returns a Boolean value indicating whether the binary is valid or not. By using this method, we can guarantee that programs are free of syntax errors or other issues that could lead to test failures. The validation process helps ensure the integrity and correctness of the WebAssembly programs being evaluated, ensuring their compliance with the WebAssembly specification.

In order to enhance the statistical significance of the results and minimize the impact of chance, we conducted three rounds of 24 h experiments. We counted the number of samples generated and the number of valid samples in each round, and the results are presented in Table 1. It is noteworthy that every sample generated by ESFuzzer successfully passed the validation process conducted using the JavaScript API function. This result indicates that ESFuzzer was successful in producing valid and syntactically correct testcases that comply with the requirements of the JavaScript API. This validation step instills confidence in the quality and reliability of the generated samples, further supporting the effectiveness of ESFuzzer in producing valid test inputs.

Table 1. Code validation results (3 × 24 h).

	Total	**Valid**	**Percent**
No. 1	960,516	960,516	100%
No. 2	957,050	957,050	100%
No. 3	975,717	975,717	100%

RQ2: Has the code coverage improved with our tool?

To assess the impact of ESFuzzer on enhancing code coverage, we utilized the llvm-cov component. Table 2 summarizes the results of the analysis, presenting the four statistics for particle size in the first four rows.

Table 2. The code coverage and crash results (3 × 24 h).

Level	Total	ESFuzzer		libFuzzer	
		Number	Percent	Number	Percent
lines	11,280	4244	37.62%	2359	20.91%
regions	17,366	4324	24.90%	1677	9.66%
branches	9152	2512	27.45%	940	10.27%
functions	908	455	50.11%	202	22.25%
Average			35.02%		15.78%

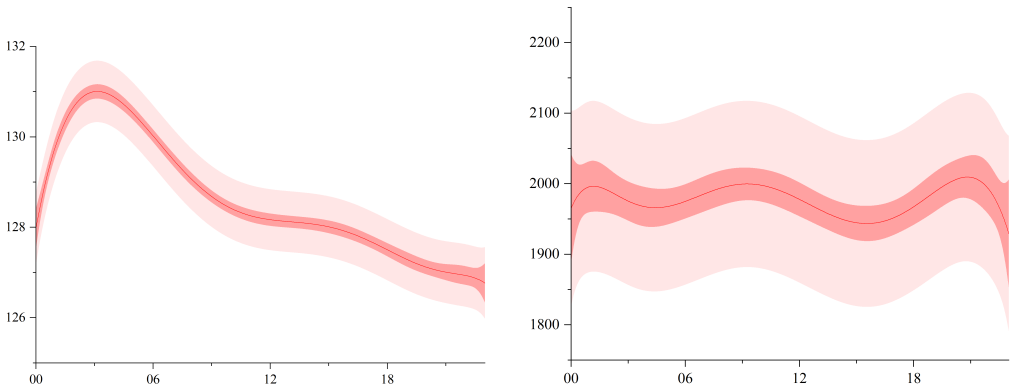
Table 2 demonstrates that ESFuzzer achieved twice the coverage advantage of LibFuzzer over three 24 h sessions in terms of code coverage across all categories, including lines, regions, branches, and functions. This indicates that ESFuzzer explored a larger portion of the codebase compared to LibFuzzer.

RQ3: Has the efficiency of fuzzing improved with the use of our tool?

To compare the efficiency of the two fuzzers, we assessed their average sample size and execution rate in the experiment. These metrics provide insights into the performance and resource utilization of the fuzzing processes.

The average sample size indicates the size of the testcases or inputs generated and used by the tools during the experiment. It serves as an indicator of the complexity and diversity of the test inputs generated by each tool. A larger average sample size indicates that the tool has explored a broader range of potential inputs, which could potentially result in better code coverage and bug detection.Therefore, we calculated the average sample size hourly and present the results in Appendix A.

Figure 3 illustrates the variation in the average sample size over time. The results for libFuzzer (Figure 3a) indicate an initial increase of 131.4 bytes per second, followed by a gradual decrease. This decline can be attributed to the fact that after 4 h, the samples generated by libFuzzer were no longer able to trigger new paths in the WebAssembly interpreter, and therefore could not generate any more complex code.



(**a**) libFuzzer (**b**) ESFuzzer

Figure 3. The average sample size during 24 h of fuzzing on the x-axis, with the corresponding values plotted on the y-axis. The graph displays the fitted curve with confidence limits (darker pink) and prediction limits (lighter pink) for the 24 h period.

In contrast, ESFuzzer (see Figure 3b) outperformed libFuzzer, achieving an average sample size change of approximately 1980 bytes per second. ESFuzzer's mutation strategy was designed to maintain the stability of the average sample size, ensuring the complexity of the samples and the possibility of triggering new paths.

The execution rate measures the speed at which the tools execute testcases. It quantifies the number of testcases processed or executed per unit of time. A higher execution rate

indicates greater efficiency in processing testcases, enabling more thorough exploration of the target system within a given time frame.

Table 3 compares the efficiency of libFuzzer and ESFuzzer based on the average sample size and execution rate metrics.

The table shows that libFuzzer generated 0.17 GB, 0.16 GB, and 0.19 GB of valid samples in three separate measurements, while ESFuzzer generated significantly larger average sample sizes of 1.77 GB, 1.76 GB, and 1.81 GB. The variation in the valid sample size between the two tools can be attributed to ESFuzzer achieving a larger average sample size compared to libFuzzer. This suggests that ESFuzzer explored a broader range of potential inputs, potentially resulting in improved code coverage and bug detection.

Table 3. The total sample size and execution rate of libFuzzer and ESFuzzer, measured in gigabytes (GB) and bytes per second (B/s), respectively.

	Total Sample Size (GB)		**Exec Rate (B/s)**	
	ESFuzzer	**libFuzzer**	**ESFuzzer**	**libFuzzer**
No. 1	1.77	0.17	21,939	2120
No. 2	1.76	0.16	21,833	1954
No. 3	1.81	0.19	22,571	2327
Average	1.78	0.17	22,114	2133

Table 3 shows that libFuzzer had execution rates of only 2120 bytes, 1954 bytes, and 2327 bytes per second in the three experimental runs. In contrast, ESFuzzer achieved much higher execution rates of 21,939 bytes, 21,833 bytes, and 22,571 bytes per second. These results suggest that ESFuzzer outperformed libFuzzer in terms of execution speed, allowing for more efficient processing of testcases and exploration of the target system within a specified time frame.

Overall, the results presented in Table 3 indicate that ESFuzzer outperformed libFuzzer in terms of sample quality, as demonstrated by the larger average sample size and execution rate. ESFuzzer generated significantly larger valid samples and demonstrated faster execution rates, indicating improved performance and efficiency compared to libFuzzer.

7. Conclusions

This paper presents ESFuzzer as a solution to the challenges of fuzzing WebAssembly interpreters. The first challenge addressed is the inefficiencies in the execution of fuzzing tests caused by WebAssembly's static checking mechanism. The second challenge addressed is the low coverage problems caused by the lack of a targeted mutation strategy.

This paper proposes an algorithm called the Stack Repair Algorithm that can effectively address the challenges of WebAssembly's interpreter static validation mechanism. Additionally, three mutation strategies based on the concepts of Equivalent-Statement and Equivalent-Exchange are designed to increase the complexity of testcases and improve code coverage. The evaluation results demonstrate that the proposed approach outperforms libFuzzer in several aspects.

Although our scheme effectively addresses the first challenge, it only partially addresses the second challenge and does not completely solve the problem. Therefore, future research should focus on improving the efficiency of fuzzing WebAssembly interpreters by increasing code coverage.

Author Contributions: Conceptualization, J.H. and X.C.; Methodology, J.H.; Supervision, Z.Z. and Y.D.; Writing—original draft, J.H.; Writing—review and editing, Z.Z. and W.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Key Research and Development Program of China [Grant No. 2017YFB0803000] and the National Natural Science Foundation of China [Grant No. U1836117].

Data Availability Statement: All data underlying the results are available as part of the article and no additional source data are required.

Conflicts of Interest: Author Yuejin Du was employed by the company Beijing Qihoo Technology Co., Ltd. and Xiuyuan Chen was employed by the company Chengdu Junzheng Technology Co., Ltd. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Appendix A

To enhance the statistical significance of the experimental results and minimize the impact of chance, we conducted three 24 h rounds of experiments and calculated the average sample size generated per hour. This approach enabled us to better comprehend the variability and randomness inherent in the experiment, aiding in the determination of the authenticity of observed effects and the evaluation of the experimental results' credibility. After conducting experiments for 24 h, variations in the mean at different time points could be observed, as shown in Table A1.

Table A1. The average sample size was measured during the fuzzing test using both libFuzzer and ESFuzzer.

	ESFuzzer			**libFuzzer**		
	No. 1	**No. 2**	**No. 3**	**No. 1**	**No. 2**	**No. 3**
0:00	2010.19	1988.26	1947.13	129.01	128.85	130.78
1:00	1951.22	2045.11	1974.76	129.85	129.83	130.03
2:00	1890.35	1955.71	2005.04	130.37	130.50	130.29
3:00	2081.00	1969.34	1961.12	131.14	131.23	131.24
4:00	2067.74	1989.52	1986.56	131.04	131.03	131.05
5:00	1990.67	1878.36	1894.70	130.10	130.35	130.14
6:00	2008.66	1975.41	1906.67	129.74	129.98	129.99
7:00	1988.79	1999.17	2031.58	129.10	129.29	129.26
8:00	1987.29	2068.14	1895.87	128.75	129.00	128.95
9:00	2073.54	1989.51	1985.09	128.60	128.62	128.60
10:00	2027.47	2009.11	2086.80	128.36	128.50	128.37
11:00	1903.81	2058.28	1951.70	128.31	128.36	128.36
12:00	1840.19	2038.37	2004.77	128.17	128.25	128.22
13:00	1916.38	1959.31	2042.05	128.07	128.16	128.15
14:00	1893.98	1911.15	2005.71	128.00	128.01	128.01
15:00	1996.16	1917.37	1911.74	127.89	127.93	127.94
16:00	1907.07	1965.20	2027.69	127.77	127.82	127.83
17:00	1929.54	1980.80	1986.80	127.65	127.73	127.69
18:00	1974.43	1946.88	2023.53	127.45	127.54	127.51
19:00	2037.25	1988.92	1965.01	127.22	127.25	127.26
20:00	1960.79	2023.85	2025.14	127.01	127.07	127.05
21:00	2023.16	2017.07	2087.93	127.00	126.98	127.03
22:00	1990.75	1963.74	1932.69	126.82	126.90	126.85
23:00	1976.95	1952.01	2024.20	127.54	127.66	126.54
24:00	2001.27	2042.55	2051.16	126.58	127.02	127.13

References

1. The Problems and Promise of WebAssembly. Available online : https://googleprojectzero.blogspot.com/2018/08/the-problems-and-promise-of-webassembly.html (accessed on 6 April 2024).
2. Wang, S.; Yuan, Y.; Wang, X.; Li, J.; Qin, R.; Wang, F.Y. An overview of smart contract: Architecture, applications, and future trends. In Proceedings of the 2018 IEEE Intelligent Vehicles Symposium (IV), Changshu, China, 26–30 June 2018; pp. 108–113.
3. Li, J.; Zhao, B.; Zhang, C. Fuzzing: A survey. *Cybersecurity* **2018**, *1*, 6. [CrossRef]
4. American Fuzzy Lop. Available online: http://lcamtuf.coredump.cx/afl (accessed on 10 April 2024).
5. Libfuzzer: A Library for Coverage-Guided Fuzz Testing. Available online: https://llvm.org/docs/LibFuzzer.html (accessed on 10 April 2024).
6. Yang, X.; Chen, Y.; Eide, E.; Regehr, J. Finding and understanding bugs in C compilers. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, San Jose, CA, USA, 4–8 June 2011; pp. 283–294.

7. MozillaSecurity/Funfuzz: A Collection of Fuzzers in a Harness for Testing the SpiderMonkey JavaScript Engine. Available online: https://github.com/MozillaSecurity/funfuzz (accessed on 11 April 2024).
8. Lindig, C. Random testing of C calling conventions. In Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging, Monterey, CA, USA, 19–21 September 2005; pp. 3–12.
9. Groß, S.; Koch, S.; Bernhard, L.; Holz, T.; Johns, M. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In Proceedings of the Network and Distributed Systems Security (NDSS) Symposium, San Diego, CA, USA, 27 February–3 March 2023.
10. Han, H.; Oh, D.; Cha, S.K. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In Proceedings of the NDSS, San Diego, CA, USA, 24–27 February 2019.
11. He, X.; Xie, X.; Li, Y.; Sun, J.; Li, F.; Zou, W.; Liu, Y.; Yu, L.; Zhou, J.; Shi, W.; et al. Sofi: Reflection-augmented fuzzing for javascript engines. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual, 15–19 November 2021; pp. 2229–2242.
12. Park, S.; Xu, W.; Yun, I.; Jang, D.; Kim, T. Fuzzing javascript engines with aspect-preserving mutation. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; pp. 1629–1642.
13. Holler, C.; Herzig, K.; Zeller, A. Fuzzing with Code Fragments. In Proceedings of the USENIX Security Symposium, Bellevue, WA, USA, 8–10 August 2012; pp. 445–458.
14. Veggalam, S.; Rawat, S.; Haller, I.; Bos, H. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *Computer Security—Proceedings of the ESORICS 2016: 21st European Symposium on Research in Computer Security, Heraklion, Greece, 26–30 September 2016*; Proceedings, Part I 21; Springer: Berlin/Heidelberg, Germany, 2016; pp. 581–601.
15. Wang, J.; Chen, B.; Wei, L.; Liu, Y. Superion: Grammar-aware greybox fuzzing. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 724–735.
16. Van Sprundel, I. Fuzzing: Breaking software in an automated fashion. In Proceedings of the 22C3 Chaos Communication Congress, Berlin, Germany, 27–30 December 2005; p. 16.
17. Stephens, N.; Grosen, J.; Salls, C.; Dutcher, A.; Wang, R.; Corbetta, J.; Shoshitaishvili, Y.; Kruegel, C.; Vigna, G. Driller: Augmenting fuzzing through selective symbolic execution. In Proceedings of the NDSS, San Diego, CA, USA, 21–24 February 2016; Volume 16, pp. 1–16.
18. Aschermann, C.; Frassetto, T.; Holz, T.; Jauernig, P.; Sadeghi, A.R.; Teuchert, D. NAUTILUS: Fishing for Deep Bugs with Grammars. In Proceedings of the NDSS, San Diego, CA, USA, 24–27 February 2019.
19. Peach Tech/Peach-Fuzzer-Community. Available online: https://gitlab.com/peachtech/peach-fuzzer-community (accessed on 11 April 2024).
20. Viide, J.; Helin, A.; Laakso, M.; Pietikäinen, P.; Seppänen, M.; Halunen, K.; Puuperä, R.; Röning, J. Experiences with Model Inference Assisted Fuzzing. *WOOT* **2008**, *2*, 1–2.
21. Manès, V.J.; Han, H.; Han, C.; Cha, S.K.; Egele, M.; Schwartz, E.J.; Woo, M. The art, science, and engineering of fuzzing: A survey. *IEEE Trans. Softw. Eng.* **2019**, *47*, 2312–2331. [CrossRef]
22. Bastani, O.; Sharma, R.; Aiken, A.; Liang, P. Synthesizing program input grammars. *ACM SIGPLAN Not.* **2017**, *52*, 95–110. [CrossRef]
23. Höschele, M.; Zeller, A. Mining input grammars from dynamic taints. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, 3–7 September 2016; pp. 720–725.
24. Honggfuzz. Available online: https://honggfuzz.dev/ (accessed on 11 April 2024).
25. Fioraldi, A.; Maier, D.; Eißfeldt, H.; Heuse, M. AFL++ combining incremental steps of fuzzing research. In Proceedings of the 14th USENIX Conference on Offensive Technologies, Berkeley, CA, USA, 11 August 2020; p. 10.
26. Gan, S.; Zhang, C.; Qin, X.; Tu, X.; Li, K.; Pei, Z.; Chen, Z. Collafl: Path sensitive fuzzing. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 21–23 May 2018; pp. 679–696.
27. Kersten, R.; Luckow, K.; Păsăreanu, C.S. POSTER: AFL-based Fuzzing for Java with Kelinci. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017; pp. 2511–2513.
28. Böhme, M.; Pham, V.T.; Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 1032–1043.
29. Chen, P.; Chen, H. Angora: Efficient fuzzing by principled search. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 21–23 May 2018; pp. 711–725.
30. Introducing Jsfunfuzz. Available online: http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz (accessed on 11 April 2024).
31. Project Zero: The Great DOM Fuzz-Off of 2017. Available online: https://googleprojectzero.blogspot.com/2017/09/the-great-dom-fuzz-off-of-2017.html (accessed on 11 April 2024).
32. Padhye, R.; Lemieux, C.; Sen, K.; Papadakis, M.; Le Traon, Y. Semantic fuzzing with zest. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Beijing, China, 15–19 July 2019; pp. 329–340.
33. Guo, R. MongoDB's JavaScript Fuzzer: The fuzzer is for those edge cases that your testing didn't catch. *Queue* **2017**, *15*, 38–56. [CrossRef]
34. Wang, J.; Chen, B.; Wei, L.; Liu, Y. Skyfire: Data-driven seed generation for fuzzing. In Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2017; pp. 579–594.

35. posidron/dharma: Generation-Based, Context-Free Grammar Fuzzer. 2022. Available online: https://github.com/posidron/dharma (accessed on 11 April 2024).
36. Gopinath, R.; Zeller, A. Building fast fuzzers. *arXiv* **2019**, arXiv:1911.07707.
37. Rawat, S.; Jain, V.; Kumar, A.; Cojocar, L.; Giuffrida, C.; Bos, H. Vuzzer: Application-aware evolutionary fuzzing. In Proceedings of the NDSS, San Diego, CA, USA, 26 February–1 March 2017; Volume 17, pp. 1–14.
38. Claessen, K.; Hughes, J. QuickCheck: A lightweight tool for random testing of Haskell programs. In Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, Montreal, QC, Canada, 18–21 September 2000; pp. 268–279.
39. Lee, S.; Han, H.; Cha, S.K.; Son, S. Montage: A neural network language model-guided javascript engine fuzzer. In Proceedings of the 29th USENIX Conference on Security Symposium, Boston, MA, USA, 12–14 August 2020; pp. 2613–2630.
40. McFadden, B.; Lukasiewicz, T.; Dileo, J.; Engler, J. Security Chasms of Wasm. In NCC Group Whitepaper. 2018. Available online: https://git.edik.cn/book/awesome-wasm-zh/raw/commit/e046f91804fb5deb95affb52d6348de92c5bd99c/spec/us-18-Lukasiewicz-WebAssembly-A-New-World-of-Native_Exploits-On-The-Web-wp.pdf (accessed on 11 April 2024).
41. Lehmann, D.; Kinder, J.; Pradel, M. Everything old is new again: Binary security of webassembly. In Proceedings of the 29th USENIX Conference on Security Symposium, Boston, MA, USA, 12–14 August 2020; pp. 217–234.
42. Lehmann, D.; Torp, M.T.; Pradel, M. Fuzzm: Finding memory bugs through binary-only instrumentation and fuzzing of webassembly. *arXiv* **2021**, arXiv:2110.15433.
43. Konoth, R.K.; Vineti, E.; Moonsamy, V.; Lindorfer, M.; Kruegel, C.; Bos, H.; Vigna, G. Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018, pp. 1714–1730.
44. Haßler, K.; Maier, D. Wafl: Binary-only webassembly fuzzing with fast snapshots. In Proceedings of the Reversing and Offensive-Oriented Trends Symposium, Vienna, Austria, 18–19 November 2021; pp. 23–30.
45. Huang, Y.; Jiang, B.; Chan, W.K. EOSFuzzer: Fuzzing eosio smart contracts for vulnerability detection. In Proceedings of the 12th Asia-Pacific Symposium on Internetware, Singapore, 1–3 November 2020; pp. 99–109.
46. Maier, D.C. Automated Security Testing of Unexplored Targets Through Feedback-Guided Fuzzing. Ph.D. Thesis, Technische Universität Berlin, Berlin, Germany, 2023.
47. dwfault/afl-wasm: forked from afl. Available online: https://github.com/dwfault/afl-wasm (accessed on 11 April 2024).
48. Wasm-Fuzzer. Available online: https://github.com/KTH/slumps/tree/master/wasm-fuzzer (accessed on 11 April 2024).
49. Fuzzing JavaScript WebAssembly APIs Using Dharma/Domato (V8 Engine). Available online: https://fuzzinglabs.com/fuzzing-javascript-wasm-dharma-chrome-v8/ (accessed on 11 April 2024).
50. Chen, W.; Sun, Z.; Wang, H.; Luo, X.; Cai, H.; Wu, L. Poster: Uncovering Vulnerabilities in Wasm Smart Contracts. In Proceedings of the 2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS), Hong Kong, China, 18–21 July 2023; pp. 1073–1074.
51. Li, W.; Wang, M.; Yu, B.; Shi, Y.; Fu, M.; Shao, Y. Grey-box Fuzzing Based on Execution Feedback for EOSIO Smart Contracts. In Proceedings of the 2022 29th Asia-Pacific Software Engineering Conference (APSEC), Virtual, 6–9 December 2022; pp. 1–10.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.