WALTZZ: WebAssembly Runtime Fuzzing with Stack-Invariant Transformation

Lingming Zhang[1], Binbin Zhao[1,2,4], Jiacheng Xu[1], Peiyu Liu[1], Qinge Xie[2], Yuan Tian[3], Jianhai Chen[1], and Shouling Ji[1*]

[1]*Zhejiang University,* [2]*Georgia Institute of Technology,* [3]*UCLA,* [4]*Engineering Research Center of Blockchain Application, Supervision And Management (Southeast University), Ministry of Education*

Abstract

WebAssembly (Wasm) is a binary instruction format proposed by major browser vendors to achieve near-native performance on the web and other platforms. By design, Wasm modules should be executed in a memory-safe runtime, which acts as a trusted computing base. Therefore, security vulnerabilities inside runtime implementation can have severe impacts and should be identified and mitigated promptly.

Fuzzing is a practical and widely adopted technique for uncovering bugs in real-world programs. However, to apply fuzzing effectively to the domain of Wasm runtimes, it is vital to address two primary challenges: (1) Wasm is a stack-based language and runtimes should verify the correctness of stack semantics, which requires fuzzers to meticulously maintain desired stack semantics to reach deeper states. (2) Wasm acts as a compilation target and includes hundreds of instructions, making it hard for fuzzers to explore different combinations of instructions and cover the input space effectively.

To address these challenges, we design and implement WALTZZ, a practical greybox fuzzing framework tailored for Wasm runtimes. Specifically, WALTZZ proposes the concept of stack-invariant code transformation to preserve appropriate stack semantics during fuzzing. Next, WALTZZ introduces a versatile suite of mutators designed to systematically traverse diverse combinations of instructions in terms of both control and data flow. Moreover, WALTZZ designs a skeleton-based generation algorithm to produce code snippets that are rarely seen in the seed corpus. To demonstrate the efficacy of WALTZZ, we evaluate it on seven well-known Wasm runtimes. Compared to the state-of-the-art works, WALTZZ can surpass the nearest competitor by finding 12.4% more code coverage even within the large code bases and uncovering 1.38× more unique bugs. Overall, WALTZZ has discovered 20 new bugs which have all been confirmed and 17 CVE IDs have been assigned.

* Shouling Ji is the corresponding author.

1 Introduction

Historically, JavaScript (JS) has been the sole programming option available on the web platform. However, it runs into severe performance problems when handling computation-intensive tasks. In response, WebAssembly (Wasm) [23] has been introduced by four major browser vendors as a *portable*, *safe*, *low-level* code format for efficient execution. Wasm can be viewed as a virtual instruction set architecture and thereby serves as a compilation target for programming languages like C++. For instance, C++ source files can be compiled into Wasm modules by Emscripten [64] and run on the web with a near-native performance.

Wasm modules are typically executed in a memory-safe, sandboxed host environment. Initially, only web browsers incorporated such environments. However, with the increasing popularity of Wasm, standalone runtimes have gradually emerged, making Wasm more versatile and applicable beyond mere web scenarios [59]. As every coin has two sides, the growing popularity of Wasm also brings more security threats. Wasm is designed with safety as a priority, host security is thereby of paramount importance for Wasm. Breaking the host security can result in severe outcomes including sandbox escape and remote code execution, e.g., CVE-2024-2887 in V8 [39] and CVE-2021-30734 in JavaScriptCore [43] are two vulnerabilities in Wasm host environments, which have been further exploited in Pwn2Own [12] to achieve remote code execution. Therefore, it is imperative to identify and rectify security bugs in Wasm runtimes promptly to prevent further malicious exploits.

Fuzzing is one of the most effective techniques for finding vulnerabilities, which generates a large amount of test cases and feeds them to the target program to detect any unexpected behaviors. Unlike static analysis techniques such as symbolic execution, fuzzing scales effectively to larger programs and is basically free from false positives. Fuzzers can be broadly classified into two categories: *mutational* and *generative*. Mutational fuzzers generate test cases by modifying existing seed inputs. For instance, AFL [20] applies byte-level mutations to

seed inputs and adds a test case to the seed pool if it uncovers new code paths, which has identified thousands of bugs across numerous software systems. Conversely, generative fuzzers create test cases from scratch based on predefined rules. For example, wasm-smith [10] takes a random number as input and produces a corresponding Wasm module, which has been integrated into runtimes like SpiderMonkey.

Despite the variety of techniques proposed for enhancing the effectiveness of fuzzing [26, 33, 35, 48], *automatic Wasm runtime fuzzing remains an underexplored area*. Effectively applying fuzzing to the realm of Wasm runtimes is still non-trivial due to the following challenges.

Challenge I: Maintenance of appropriate stack semantics. Wasm is a stack-based language where instructions are conceptualized as operations that pop and push stack values of specific types. Wasm runtimes are demanded to validate the correctness of stack semantics of input modules, allowing only valid modules to be instantiated and executed. However, it is challenging for mutational fuzzers to maintain desired stack semantics during mutation since even a single byte flip can alter the interpretation of stack semantics, hindering fuzzer's ability to explore deeper runtime states.

Challenge II: Effective exploration of input space. Given a fixed fuzzing period, it is essential to cover as much input space as possible. However, since Wasm comprises hundreds of instructions aimed for varied purposes, it is difficult for fuzzers to traverse different combinations of instructions comprehensively and effectively. This particularly affects generative fuzzers, as their exploration of the input space is severely constrained by predefined rules.

With regard to these challenges, we propose WALTZZ, an effective greybox fuzzing framework designed specifically for Wasm runtimes. To solve **Challenge I**, we propose *stack-invariant transformation* to preserve stack semantics during fuzzing. Our key observation is that one instruction sequence can be interchanged seamlessly with another if their effects on stack semantics are compatible. This conclusively guarantees the validity of generated inputs and serves as the core guiding principle of the entire design. To tackle **Challenge II**, we devise a suite of Wasm mutators that synergistically enhance the diversity of control and data flow in the input. Moreover, we design a *skeleton-based generation* algorithm that isolates the generation of control flow skeleton from the production of actual data flow, which enables the construction of code snippets that are rarely encountered in the seed corpus.

We have implemented WALTZZ and evaluated it on seven Wasm runtimes: JavaScriptCore, SpiderMonkey, V8, wasm3, wasm-interp, wasmtime, and wasm-micro-runtime. Compared to state-of-the-art works such as AFL++, WALTZZ achieves superior performance on code coverage, outperforming the nearest competitor by 12.4% even within the sizeable code bases. In respect of the bug-finding capacity, WALTZZ discovers 1.38× more unique bugs than the second best. Overall, WALTZZ has detected 20 previously unknown bugs. All of them have been confirmed, with 17 assigned CVE IDs. While many of these bugs have been fixed promptly, others demand further structural modifications to the code base and we discuss one of them in this study. This highlights the critical nature of bugs found by WALTZZ.

In summary, this paper makes the following contributions:

- We propose WALTZZ, a practical and effective greybox fuzzing framework tailored particularly for Wasm runtimes. Our approach incorporates a novel stack-invariant transformation technique to maintain appropriate stack semantics throughout fuzzing. The code transformation leverages the insight that instructions with compatible stack semantics could be exchanged without violating validity checks.
- Based on the stack-invariant transformation, we design a rich set of Wasm mutators that augment both control and data flow diversity in seed inputs, enabling WALTZZ to explore the input space more effectively. We also present a novel skeleton-based generation algorithm to produce rarely-seen code snippets.
- We have implemented WALTZZ and evaluated it on seven Wasm runtimes. Compared to the state-of-the-art works, WALTZZ demonstrates its superiority in achieving 12.4% more code coverage and discovering 1.38× more unique bugs than the second best. In total, WALTZZ has found 20 new bugs, all of which have been verified, with 17 received CVE IDs.

To foster further research on this topic, we have made the prototype implementation of WALTZZ available at https://zenodo.org/records/14718828 and have provided a more detailed collection of the artifacts at https://github.com/mobsceneZ/Waltzz.

2 Background

In this section, we begin by introducing the file structure of Wasm, followed by an overview of its type system and the validity check mechanism.

2.1 Wasm File Structure

As illustrated in Figure 1a, the Wasm module serves as the unit of deployment, loading, and compilation, which contains the following main components:

- **Table**: Table section declares any number of tables, and is used to hold opaque values like function references;
- **Memory**: Memory section defines a randomly accessible linear array to store raw uninterpreted bytes;
- **Data Segment**: Data segments can be used to initialize a range of memory from a static vector of bytes;

- **Element Segment**: Similarly, element segments can be utilized to initialize a specific range of the table using a static vector of elements;
- **Global**: Globals act as module-wide locals that can be declared as mutable or immutable and further provided with initialization values;
- **Function**: Function section comprises a vector of functions, each of which defines the function signature, types of the local variables, and the function body;
- **Import/Export**: The Wasm module can import (export) functions, tables, memory, and globals from (to) the host environment;

In essence, the Wasm module can be regarded as a collection of instruction sequences, accompanied by surrounding contexts such as memory to support specific operations. This forms the foundation for the validity check.

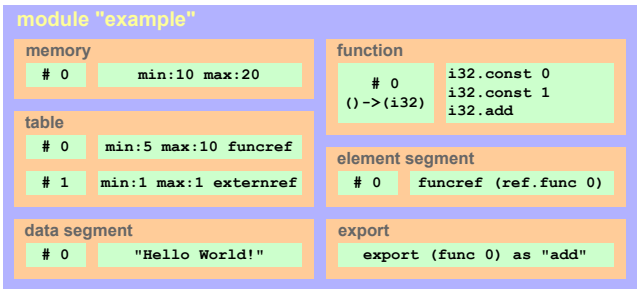
2.2 Wasm Validity Check

According to the core specification [53], a Wasm module must satisfy all validity checks to ensure its well-formedness. Only valid modules can be instantiated and executed.

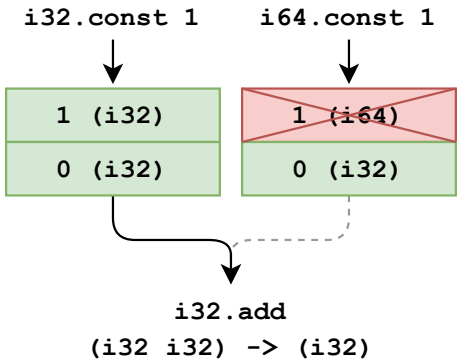
Specifically, validity is intricately linked to the underlying type system. In Wasm, there exist seven distinct primitive types: `i32`, `i64`, `f32`, `f64`, `v128`, `funcref`, and `externref`. An `externref` type denotes a reference to an object owned by the host environment. For a Wasm module to be valid, all instructions should be type-checked for their stack semantics. For simplicity, we consider the operation of popping certain value types $[t_1^*]$ from the stack and pushing new value types $[t_2^*]$ back as instruction's stack semantics, denoted by the stack type $[t_1^*] \rightarrow [t_2^*]$.

Consider an instruction s_i in the sequence $[s_0, ..., s_{i-1}, s_i, ...]$ with stack type $[t_1^*] \rightarrow [t_2^*]$, a *necessary condition* for s_i to be type-checked is that the preceding sub-sequence $[s_0, ..., s_{i-1}]$ should generate $[t_3^* \; t_1^*]$ on the stack, thereby allowing $[t_1^*]$ to be correctly consumed by s_i. Figure 1b demonstrates an example of the type-checking process. The stack type of `i32.add` is $[i32 \; i32] \rightarrow [i32]$, which pops two `i32` values and pushes the sum back. Thus, `i32.add` can be preceded by `i32.const` but not `i64.const` even if the pushed values are identical, as the latter one produces an `i64` type.

For most instructions, the necessary condition mentioned above is also sufficient. However, some instructions interact with the surrounding contexts, thereby demanding additional validity checks. For example, variable instructions require locals or globals to be present. Note that the stack semantics of these instructions might vary according to the context, e.g., `global.get` will push a value matching the type of relevant global variable. More detailed validity check mechanisms are elaborated in later sections as necessary.



(a) A demonstrative example of the Wasm file structure.



(b) The type-checking process of `i32.add`.

Figure 1: Wasm modules are composed of different parts such as data segments, as subfigure (a) shows. The validity check mainly ensures that every instruction is type-checked and the subfigure (b) presents the type-checking process of `i32.add` instruction.

3 Motivation

Currently, mutational fuzzers are predominant in both academic and industrial settings. While considerable efforts have been committed to enhancing the effectiveness of mutational fuzzers, many approaches still share similar mutation strategies, namely, byte-level mutations. For instance, the state-of-the-art greybox fuzzer AFL++ [16], might perform a simple bit-flip mutation, converting the value `0x41` to `0x43` within a seed. In file formats such as PNG, such a minor modification generally has minimal impact. However, the situation is markedly different for Wasm. A substantial portion of the input bytes are carried with stack semantics, e.g., `0x41` and `0x43` represent the opcodes for `i32.const` and `f32.const` respectively, each operating on different types of stack values. Since Wasm runtimes ensure that input modules must adhere to appropriate stack semantics, such a small mutation on valid inputs can immediately corrupt the stack states and result in early rejection by the runtime.

Therefore, it is difficult for current mutational fuzzers to effectively test Wasm runtimes, as they struggle to maintain desired stack semantics during each mutation step. To further consolidate this argument, we have run the representa-

tive mutational fuzzer AFL++ on three JS engines, i.e., V8, JavaScriptCore, and SpiderMonkey, for 24 hours and repeated 10 times. The initial seed corpus is extracted from the official test suite [52] and wasm-validate from the Wasm Binary Toolkit [50] is utilized to verify the validity of the generated test cases. Table 1 summarizes the results, showing that over 98% of the test cases generated by AFL++ are invalid. This demonstrates the inefficacy of stack-ignorant mutations, leading directly to **Challenge I**, which our work aims to address.

Another line of work adopts a generation-based approach to construct valid inputs from scratch, such as wasm-smith [10]. Generative fuzzers typically depend on random sources and predefined rules to dictate the appearance of generated test inputs. However, given that Wasm is an instruction set with rich semantics, it is challenging for generative fuzzers to cover the input space thoroughly, as the possible exploration space is significantly constrained by hand-written rules. For instance, wasm-smith uses a generation strategy that randomly selects instructions consistent with the current stack state for generation. While ensuring the validity of produced modules, this approach significantly hinders the generation of certain instructions, e.g., `v128.bitselect` requires wasm-smith to produce three `v128` values on top of the stack before it can be selected for generation. Consequently, the input space that generative fuzzers explore is relatively constrained, which directly relates to **Challenge II** that our work seeks to solve.

To achieve the best of both worlds, we design WALTZZ with a thorough consideration of these challenges. Next, we explain how we address these challenges throughout the design, implementation, and evaluation of WALTZZ.

Table 1: The average counts of total and valid test cases from fuzzing three major JS engines across 10 runs, each with a 24-hour budget using AFL++.

Project	$Num_{Testcase}$	Num_{Valid}	$\frac{Num_{Valid}}{Num_{Testcase}}$
V8	793041.7	7657.9	0.9656%
JavaScriptCore	241337.1	3894.4	1.6137%
SpiderMonkey	462344.5	16581.2	3.5863%
Avg.	498907.8	9377.8	1.8797%

4 Waltzz Design

We present WALTZZ, an effective greybox fuzzer specifically designed for Wasm runtimes. Figure 2 illustrates the overall design and workflow of WALTZZ. At a high level, WALTZZ takes an instrumented Wasm runtime, a collection of Wasm seed files, and optionally a harness as inputs, then outputs the bugs identified within the runtime. Starting with a seed input, WALTZZ parses it to construct its *typed IR* and subsequently fragments the IR into smaller *code segments*, each labeled with its corresponding stack type (① in Figure 2). The typed

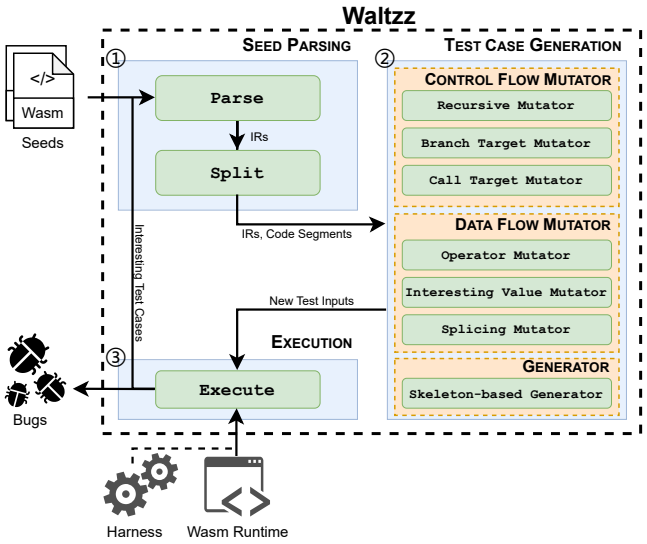


Figure 2: High-level overview of WALTZZ.

IR then undergoes a series of *stack-invariant* mutations and generations, producing a list of transformed IRs. These IRs are lifted back into Wasm modules afterward (② in Figure 2). Finally, the generated modules are sequentially executed by the runtime. If new code paths are uncovered or the runtime crashes, the corresponding test case is added to the seed pool or the crash directory respectively. Otherwise, the test case is discarded (③ in Figure 2). Note that JS engines cannot directly execute Wasm modules, therefore, a test harness that invokes relevant APIs is necessary to process Wasm inputs.

The design of WALTZZ centers around the generation of effective test cases to explore deep and broad program logic of Wasm runtimes. In Section 4.1, we elucidate the concept of stack-invariant transformation, which serves as a general guideline for the whole design and efficaciously addresses the **Challenge I**. Building on this fundamental principle, we then discuss the design of mutators tailored for navigating different combinations of control flow (see Section 4.2) and data flow (see Section 4.3). Finally, Section 4.4 outlines the design of the skeleton-based generator. The synergistic effect of the mutators and the generator empowers WALTZZ to effectively traverse the input space, thereby addressing the **Challenge II** efficiently.

4.1 Stack-Invariant Transformation

Wasm runtimes enforce validity checks on input modules, ensuring that all instructions are type-checked according to the corresponding stack semantics. However, indiscriminate modifications to input modules are likely to corrupt the stack states, blocking fuzzers from reaching deep states. Thus, it is of paramount importance to establish a principle that guarantees the preservation of valid stack semantics.

We propose a concise and practical strategy termed stack-

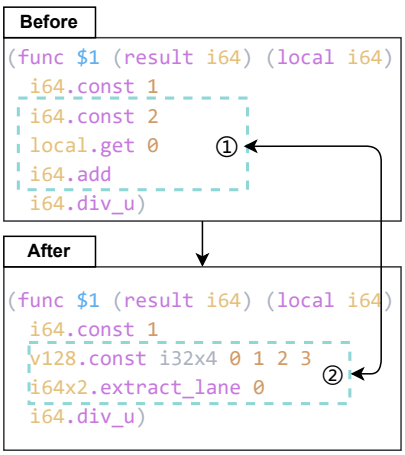


Figure 3: An example of stack-invariant transformation.

invariant transformation to address this challenge. The basic intuition is that when analyzing a sequence of instructions, we can disregard the specific operations performed on stack values and instead concentrate exclusively on the cumulative effects on the stack types, as this is the primary aspect checked by Wasm runtimes. Therefore, one instruction sequence can be interchanged with another provided that their impacts on stack types are compatible, thereby maintaining the invariant state of the stack. Figure 3 demonstrates an example of stack-invariant transformation where segments ① and ② do not consume any stack values but each produces one `i64` value to the stack. While the specific value on the stack may vary, both segments maintain the same stack type $[\,] \rightarrow [i64]$. Since the subsequent instructions still retrieve values of the expected types from the stack consistently, the modified code remains valid.

The stack-invariant transformation can flawlessly preserve desired stack semantics without imposing any requirements on the code segments undergoing transformation. Therefore, it serves as the core guiding principle throughout the design of the mutators and the generator, which are illustrated in later sections.

4.2 Control Flow Mutators

The control flow mutators are dedicated to exploring various possibilities within both intra- and inter-procedural control flow. However, directly modifying the control flow is a non-trivial task, since Wasm implements structured control flow, allowing code execution to be directed only towards specific labels and enforcing type checks on the control flow integrity.

To solve this, we adjust the stack-invariant transformation according to the surrounding contexts. Next, we devise three control flow mutators to enhance the diversity of control flow: the recursive and branch target mutator (see Section 4.2.1 and 4.2.2) alter the control structure within the function, and the call target mutator (see Section 4.2.3) modifies the calling relationships among functions.

4.2.1 Recursive Mutator

The `block`, `loop`, and `if..else` structured instructions form the basic control structure of a function, serving as containers for other instructions. All three instructions are declared with stack types $[t_1^*] \rightarrow [t_2^*]$ and are type-checked if and only if the contained instructions consume exactly $[t_1^*]$ values from the stack and produce $[t_2^*]$ values back.

The **recursive mutator** is designed to create more deeply nested control structures, which has proven useful in previous studies [3, 6]. We randomly select a control structure inside the function and encapsulate it within an additional structure. However, the control flow structure should offer equivalent stack semantics to the surrounding context before and after nesting. We address this by ensuring that the outer structure mirrors the stack type of the inner structure, effectively creating a form of self-replication.

Example 1. *Given the following randomly selected structured instruction*: `block (result i32) ... end`, *the recursive mutator could encapsulate the* `block` *with another* `loop` *structure*: `loop $0 (result i32) (block (result i32) ... end) end`. *Externally, both code snippets provide an* `i32` *type to the stack, so the transformed input remains valid.*

4.2.2 Branch Target Mutator

In Wasm, control flow redirection is achieved through the use of branch instructions. `br` and `br_if` execute unconditional and conditional jumps to a label respectively, and `br_table` performs an indirect branch through an operand indexing into a label vector.

The **branch target mutator** is devised to modify the target label of branch instructions to execute different control flows. However, unlike traditional instruction sets, labels in Wasm are accompanied by a stack type denoted as $[t^*]$ that needs to be satisfied when executing the jump. This requires additional consideration of the label context during the stack-invariant transformation. Specifically, the branch target mutator selects a target label `L1` at random and replaces it with another label `L2` that is also randomly chosen from the surrounding context. Suppose `L1` and `L2` demand $[t^*]$ and $[t'^*]$ respectively, branch target mutator employs the following adjustments:

- As `br` and `br_table` execute unconditional jumps, the stack types of these instructions are unconstrained and can be dynamically adjusted to meet the requirements of the surrounding context. We can safely substitute the original instructions that yield $[t^*]$ with new instructions that generate $[t'^*]$.
- `br_if` leaves the original $[t^*]$ values on the stack. Thus, direct modification of the stack values to type $[t'^*]$ will

likely result in an ill-formed stack state. We solve this through local code transformation, which involves inserting code segments before and after the `br_if` instruction: the first segment converts original values of type $[t^*]$ to new values of type $[t'^*]$, which are utilized by `br_if` to conduct a conditional jump to `L2`. The second segment then restores values from $[t'^*]$ back to $[t^*]$. This ensures the overall effect on the stack remains unchanged while the target label is altered.

Example 2. *Consider the branch instruction* `br_if L1` *and we aim to redirect the target from* `L1` *to* `L2`. *Suppose* `L1` *and* `L2` *need* `i64` *and* `i32` *on top of the stack respectively, we perform the following transformation*: ① *Use* `i32.wrap_i64` *to convert existing* `i64` *value to* `i32`; ② *The new* `i32` *value together with the unchanged condition value is consumed by the* `br_if L2`, *which produces an* `i32`; ③ *The* `i32` *value is turned back to* `i64` *using* `i64.extend_i32_s`. *The whole process yields the value of the same type as before, therefore, the resulting code is still valid.*

4.2.3 Call Target Mutator

Function calls are handled through `call` and `call_indirect` in Wasm. As the name indicates, the `call` instruction invokes a target function directly and the `call_indirect` instruction calls a function through an operand indexing into a `funcref` table. These instructions are type-checked according to the specified function signature.

The **call target mutator** modifies the target function of the `call` and `call_indirect`. To preserve appropriate stack semantics, it randomly selects another function with the same signature to serve as the new target. If no suitable functions are available, we resort to the generator to create a new function that satisfies the required signature. While the aforementioned steps suffice for the `call` instruction, they are not applicable for `call_indirect`, since the function invoked is determined at runtime. We address this by altering the corresponding table entry immediately before the `call_indirect` instruction to ensure the invocation of the new target.

Example 3. *To mutate* `(i32.const 10) (call_indirect $tbl (param) (result))`, *we find a function* `$f` *with the same signature* $[\,] \rightarrow [\,]$ *in input, update the table entry with* `table.set` *which requires one function reference and one table index on the stack, the former is given by the* `ref.func $f` *and the latter is given by the* `i32.const 10` *which aligns with the table index in* `call_indirect`. *By doing this right before the indirect call, we ensure the new target* `$f` *will be called.*

4.3 Data Flow Mutators

The data flow mutators are specifically designed to alter the stack values manipulated by instructions, as well as the way by which these values are processed. In particular, we propose three data flow mutators to augment the diversification of data generation and processing mechanisms: the operator mutator (see Section 4.3.1) primarily modifies the processing of stack values and the interesting value mutator (see Section 4.3.2) focuses on the modification of stack values, whereas the splicing (see Section 4.3.3) mutator handles both aspects.

4.3.1 Operator Mutator

Most of the Wasm instructions can be considered operators that manipulate stack values. For example, the `i32.add` pops two `i32` values from the stack, adds them together, and then pushes the resulting value back.

The **operator mutator** replaces one operator with another, following the principle of stack-invariant transformation. Operators are categorized into groups according to their stack semantics, e.g., `i32.add` and `i32.le_s` are grouped together since they share the same stack type $[i32\ i32] \rightarrow [i32]$. When given an operator from the input, the mutator replaces it with another operator selected randomly from the corresponding operator group, enabling varied operations on the same stack values.

Example 4. *The* `i32.add` *operator in the input can be safely substituted by the* `i32.le_s` *operator, which results in the stack values being processed differently, since the former is an arithmetic operator while the latter is a boolean operator.*

4.3.2 Interesting Value Mutator

In Wasm, immediate values of various types are pushed onto the stack using `const` instructions, e.g., `f64.const` pushes a `f64` immediate value to the stack. These values are further utilized for numerical computations, memory addressing, and other operations.

The **interesting value mutator** adjusts immediate values to boundary values like `INT_MAX`, which denotes the largest signed 32-bit integer. We maintain lists of interesting values for `i32`, `i64`, `f32`, and `f64`. For the `v128` type, values are constructed by combining interesting values of smaller types. For instance, a single `v128` value may be composed of four `i32` values. During the mutation process, the mutator selects an interesting value of the corresponding type at random and replaces the original immediate value with it.

Example 5. *We could change the value 5 in* `i32.const 5` *to 65535, which represents the largest unsigned 16-bit integer.*

4.3.3 Splicing Mutator

The concept of splicing different parts of two seed inputs has been broadly adopted in the domain of mutational fuzzing [3,25,40,46,56]. These mutations amalgamate diverse aspects of the seed inputs to produce more varied test cases, enabling fuzzers to explore additional code paths. For instance, Nautilus [3] has pointed out in the evaluation that "*Eventually splicing of interesting code fragments becomes by far the most*

effective mutation technique." We have therefore incorporated this strategy into the design of WALTZZ.

The **splicing mutator** directly reflects our stack-invariant transformation. Algorithm 1 illustrates the splicing mutation process. The mutator is given the typed IR of the current input S_1 along with the typed IR of the splicing input S_2. Initially, it fragmentizes S_2 into a pool of segments (see Line 1). Each segment comprises a single instruction and, recursively, all instructions whose outputs could influence the result of this instruction, effectively resembling a program slice. Segments are tagged with the corresponding stack types. For instance, `i32.clz` counts the number of leading zero bits of the `i32` value on the stack and pushes the result as `i32` back. If the original value is produced by `i32.const 0`, then `i32.clz` together with `i32.const 0` forms a segment tagged with the $[\,] \to [i32]$. The mutator then processes each function in S_1 and every instruction within those functions (see Lines 2-3). The mutator first attempts code substitution with a probability of p_{sub}. It identifies a potential closure $segm_{cur}$ for the current instruction and calculates the stack type of this segment. The mutator then searches for a segment $segm_{sub}$ in the pool that meets the stack type (see Lines 4-7). If found, it replaces the original segment $segm_{cur}$ in S_1 with the new segment $segm_{sub}$ in S_2 (see Lines 8-11). The code insertion follows a similar pattern, but the inserted segment should have a stack type of $[\,] \to [\,]$ to prevent any negative effects on the stack states (see Lines 13-19). Additionally, it is important to note that newly spliced-in instructions are guaranteed not to be replaced by subsequent substitutions in a single splicing run.

Although the aforementioned process seems intuitive, it still demands careful consideration during implementation. The most notable challenge is that many Wasm instructions interact with the context such as memory and Wasm runtimes also check the presence of such contexts during the validation phase. This necessitates the development of a context repair strategy. We address this in an on-demand manner: given the context C_1 required by the segment S and the current context C_2, we check if each context c_1 in C_1 can be satisfied by any context c_2 in C_2, if so, we adjust segment S to utilize c_2 instead of c_1, otherwise, we create a new context c_3 in C_2 and replace c_1 with c_3 in segment S to meet the requirement. For instance, if segment S contains an `i32.load` instruction which demands a memory context, we will add a new memory if it is not already present in the current input, otherwise, we utilize the existing one. We employ this on-demand context repair strategy immediately before the code substitution and insertion to fix up contexts such as memory, tables, and local variables (see Lines 9 and 16).

Example 6. *Suppose we have one code segment* `(i32.clz (i32.const 1))` *in current input and another code segment* `(i32.load (i32.const 0))` *in splicing input, as they share the same stack type* $[\,] \to [i32]$, *the splicing mutator can substitute the former with the latter and introduces a memory when necessary to satisfy the context requirement of* `i32.load`.

Algorithm 1 Splicing Mutation Algorithm

Input: S_1 and S_2: the typed IRs of the current and splicing input
Output: S_1': the typed IR of the mutated input

```
1:  segms ← fragmentize_input(S_2)
2:  for func ∈ S_1.functions do
3:      for instr ∈ func.instructions do
4:          if flip_coin(p_sub) == true then                # Substitution
5:              segm_cur ← get_one_segment(instr)
6:              type ← calculate_stack_type(segm_cur)
7:              segm_sub ← find_matching_segment(segms, type)
8:              if segm_sub is not null then
9:                  repair_context(segm_sub, S_1, S_2)
10:                 substitute_with_new(func, segm_cur, segm_sub)
11:             end if
12:         end if
13:         if flip_coin(p_ins) == true then                # Insertion
14:             segm_ins ← find_matching_segment(segms, [ ] → [ ])
15:             if segm_ins is not null then
16:                 repair_context(segm_ins, S_1, S_2)
17:                 insert_new_before(func, instr, segm_ins)
18:             end if
19:         end if
20:     end for
21: end for
```

4.4 Skeleton-Based Generator

Intuitively, it would also be beneficial to generate segments on our own, potentially yielding constructs that are difficult to produce solely through the mutational approach. However, since Wasm runtimes enforce type checks on stack semantics and control flow, which are intricately interconnected, it is hard for the generation algorithm to combine distinct aspects of the standard while simultaneously satisfying the required checks.

To address this, we propose a **skeleton-based generation** algorithm that separates the generation of the control flow skeleton from the creation of actual stack values, as shown in Algorithm 2. Specifically, the algorithm takes as input the target result type, which is the type of values expected to be generated on the stack, the current recursion depth, and the current context. It first plans the overall control flow structure and determines which part of the target result type each control structure is responsible for generating (see Lines 8-18). For instance, given the target type $[i32\ i64\ f32\ f64]$, the algorithm might select one `block` instruction to produce the $[i32\ i64]$ values, an `if` instruction that yields no output, and a `loop` instruction to generate the $[f32\ f64]$ values.

Control flow skeletons can be recursively generated up to a predefined maximum recursion depth, enabling the creation of more complex control flows. When the maximum recursion depth is reached, we begin filling in instructions that produce

Algorithm 2 Skeleton-based Generation Algorithm

Input: T: target result type; $depth$: current depth; C: current context
Output: O: generated segment with result type T

```
1:  function generate_top_level(T, depth, C)
2:      O ← [∅]
3:      lst : [t_1, t_2, ..., t_n] ← expand_type(T)
4:      if depth ≥ max_recursion_depth then
5:          O ← O ∪ generate_base_case(T, 0, C)
6:          return O
7:      end if
8:      while lst is not null do
9:          T' ← pop_and_build_subtype(lst)
10:         switch match (rand() % 3)
11:             case Block:
12:                 block_label ← get_label_name()
13:                 block_body ← generate_top_level(T', depth + 1,
                                    C ∪ { label : block_label })
14:                 O ← O ∪ Block(block_label, block_body, T')
                ...
15:         end switch
16:     end while
17:     return O
18: end function
```

stack values of the desired types (see Lines 4-7). This process is similar to the tree generation: suppose a value of type T is required, we randomly select an instruction whose result type is T and then recursively generate the values required by this instruction, this generation process terminates when we encounter instructions that require no further values or when the maximum recursion depth is reached. To facilitate the generation of all instructions specified in the Wasm standard, the generator is provided with a context C that can be updated dynamically throughout the generation process. This enables the generation of specific operations, such as branching to a newly introduced label, thereby strengthening the interplay between control flow and data flow.

Example 7. *When generating a code segment for target result type* $[i32\ f32\ f64]$, *the generator can use one* `block` *instruction for* $[i32]$ *and one* `loop` *instruction for* $[f32\ f64]$, *if we set the maximum recursion depth to 1, then the control flow skeleton is:* `block (result i32) □ end loop (result f32 f64) □ end`. *The* □ *can later be filled with instructions that produce the corresponding types. For instance, the first* □ *could be filled with the* `i32.reinterpret_f32 (f32.const 0.0)`, *which pushes an* `i32` *value on the stack.*

5 Implementation

We have developed a prototype of WALTZZ, comprising over 4K+ lines of C. WALTZZ functions as a custom mutator [1] atop AFL++ 4.08c and leverages AFL++'s existing fuzzing infrastructure, including its feedback mechanism and seed scheduling. To achieve the best of both worlds, WALTZZ still performs byte-level mutations on seed inputs but marks each seed with a flag indicating its validity. WALTZZ only conducts customized mutations and generations on valid seed inputs. Moreover, we set p_{sub} and p_{ins} to 0.1 and 0.067 respectively in the current implementation of WALTZZ.

Starting with a valid seed input, WALTZZ uses the official compiler and toolchain library Binaryen [49] to parse it into a typed IR. The number of mutations is then determined based on the performance score calculated by AFL++. Next, mutations are conducted in a stacking manner: during each mutation, a mutator is selected randomly to transform the IR, and the generator is applied as needed to produce the required constructs. Finally, the transformed IR is lifted back to a new module and executed by the target Wasm runtime.

It should be noted that Wasm is continuously evolving, with various language proposals currently under standardization, including garbage collection and threads. While some Wasm runtimes have already supported these proposals, WALTZZ presently adheres to the established Wasm standard for the sake of generality. Nevertheless, integrating new standardized proposals should be straightforward, as the stack semantics of new instructions are similarly type-checked. Besides, we do not currently propose any mechanisms to mitigate infinite loops due to diversity concerns. A preliminary experiment conducted using the settings detailed in Section 6.1 reveals an average timeout rate of 0.066%, which we deem acceptable.

6 Evaluation

To evaluate the efficacy of WALTZZ, the experiments have been strategically designed to address the following research questions:

RQ1: Can WALTZZ surpass existing state-of-the-art fuzzers in terms of both code coverage and bug-finding capacity? (see Section 6.2)

RQ2: What is the contribution of each mutator to the performance of WALTZZ? (see Section 6.3)

RQ3: How effective is WALTZZ at generating semantically valid inputs? (see Section 6.4)

RQ4: What are the practical implications of the identified vulnerabilities? (see Section 6.5)

6.1 Experimental Setup

We conduct experiments on two machines, each running 64-bit Ubuntu 20.04 LTS with Intel Xeon Gold 6248 (2.5GHz) CPUs (160 cores) and 256 GB of main memory.

Baselines. We select five prominent fuzzing methods as baselines to evaluate the effectiveness of WALTZZ: ① AFL++ [16]

(version 4.20c) represents the state-of-the-art greybox fuzzer which also serves as the basis for WALTZZ. ② RedQueen [4] leverages the relationships between input values and comparison instructions to overcome fuzzing roadblocks, we apply AFL++'s cmplog mode (with level 3) as an alternative, since it implements the same technique without imposing additional hardware support. ③ WasmFuzzer [27] (commit `1655d`) is a greybox fuzzer that employs structure-aware mutations to Wasm modules. ④ wasm-smith [10] (version 1.222.0) is a random Wasm module generator that is guaranteed to generate valid test cases. We further utilize AFL++ to track and mutate the random seeds supplied to wasm-smith, thereby achieving effects similar to coverage-guided fuzzing. ⑤ Wapplique [66] (commit `d56a2`) is a black-box fuzzer that performs splicing mutations and ensures the validity of the resultant modules.

Benchmarks. Wasm runtimes can be classified into web and non-web embeddings. In the web embeddings, we select three widely used JS engines: SpiderMonkey [37] (commit `3609b`), JavaScriptCore [2] (version 2.44.0), and V8 [21] (commit `cbc1b`). These engines power major web browsers such as Safari and Chrome. In the non-web embeddings, we choose four Wasm runtimes that have been broadly deployed in downstream such as Siemens: wasm-micro-runtime [9] (commit `b9740`), wasmtime [11] (version 28.0.0), wasm-interp [50] (commit `1471d`) and wasm3 [58] (commit `13907`). For each selected runtime, we use the latest version or commit available at the time of our evaluation. It is worth noting that while reproducing known bugs in older versions can be an effective approach for evaluating the performance of fuzzers [29], we choose not to adopt this method since Wasm is an evolving language and the historical commits containing bugs vary in their support for the Wasm standard. Configuring a replication environment for each bug and each fuzzer will be error-prone and time-consuming, especially for large projects like V8.

Seeds. Mutational fuzzers need initial seed corpora to start up. We construct this corpus by gathering Wasm files from the official Wasm standard tests [52], where each test comprises multiple Wasm modules with test assertions. From these, we extract only those valid modules that do not require imports, as most of the imports are specific to the spectest interpreter. Additionally, we modify each module by integrating a new function that replicates all test actions, e.g., a test assertion that checks the return value of a function call transforms into a corresponding function call in this new function. We export this function with the name `main` and designate it as the entry point for Wasm runtimes. For wasm-smith, we provide a generic seed comprised of various characters selected from the printable ASCII set: `ABC..XYZabc..xyz012..789!"..+`.

Configurations. We build all target runtimes with the default configuration. For testing web embeddings, we utilize the Wasm-JS APIs [54] to interact with the input modules. Figure 4 illustrates the test harness for V8. To evaluate wasmtime, we leverage afl.rs [45] for the instrumentation and testing of the Rust target. Figure 10 of Appendix presents the test harness for wasmtime. Other runtimes are tested in a similar manner but no test harness is required. To address potential infinite loops generated by fuzzers, we set the timeout of a single execution to 250 ms. Furthermore, we repeat each experiment 10 times to mitigate the randomness. More details on configurations will be provided in relevant subsections.

```
1 var rawBuffer = read(arguments[0], 'binary');
2
3 var validated = WebAssembly.validate(rawBuffer);
4 WebAssembly.compile(rawBuffer)
5   .then((mod) => WebAssembly.instantiate(mod))
6   .then((instance) => {instance.exports["main"]();});
```

Figure 4: Test harness for Google V8.

6.2 Comparison to State-of-the-Art Fuzzers

To address RQ1, we have run WALTZZ and other state-of-the-art fuzzers on seven Wasm runtimes as specified in Section 6.1 over a period of 72 hours. This aligns with the test duration of previous works [5, 31, 46]. Given that wasm-micro-runtime and wasm3 currently lack support for the SIMD proposal, we exclude related instructions and seeds when testing these two targets. Notably, although the implementation of reference-types and bulk-memory-operations proposals is incomplete in wasm3, we still incorporate these proposals for the comprehensiveness of testing.

Wapplique first constructs a fragment pool from the ingredient corpus and then performs splicing mutations 200 times on each seed within the seed corpus to generate new test cases. For fairness, we use the seed corpus described in Section 6.1 as both the seed and ingredient corpus for Wapplique. Besides, we configure wasm-smith to produce Wasm modules that import nothing and export a function named `main` to align with the settings in Section 6.1.

In this study, we leverage the edge coverage provided by AFL++'s collision-free instrumentation as a proxy for code coverage. Figure 5 illustrates the average number of edges achieved across six Wasm runtimes by different fuzzers. We present the complete results of log-on-time-scale coverage in Figure 11 of Appendix. We are not able to evaluate WasmFuzzer on V8 as the compiler version V8 relies on does not support WasmFuzzer's instrumentation pass. Consequently, Figure 5 only includes the available results.

For all Wasm runtimes except wasm-micro-runtime and wasm3, WALTZZ consistently outperforms competing fuzzers with the lower bound of its confidence interval higher than the upper bound of the confidence interval of any other fuzzer. For instance, WALTZZ identifies 17.4% more edges than its closest competitor AFL++ in SpiderMonkey even though the initial seeds have provided such high edge coverage. If we exclude the baseline coverage offered by seeds, WALTZZ can independently discover 94.4% more edges on average than

Table 2: Unique bugs identified by different fuzzers. The best result is marked in bold.

Fuzzer	wasm3	wasm-micro-runtime	wasm-interp	Others	Sum
WALTZZ	**23**	3	**7**	0	**33**
AFL++	20	1	3	0	24
RedQueen	12	0	0	0	12
wasm-smith	7	**5**	0	0	12
WasmFuzzer	11	0	1	0	12
Wapplique	0	0	0	0	0
Total	26	6	7	0	39

AFL++ across these targets. For the remaining two runtimes, WALTZZ does not demonstrate noticeable advantages. This is mainly because these targets are relatively lightweight and our seeds already provide satisfactory code coverage. Besides, some proposals are not fully supported by these runtimes but we still include them for the sake of comprehensive testing. Consequently, WALTZZ occasionally generates instructions not recognized by these runtimes, which can hinder its effectiveness.

Upon closer inspection of Figure 5, we have several interesting observations. Firstly, coverage guidance can improve the performance of wasm-smith on specific runtimes. Notably, it achieves steady coverage improvement and approaches the performance of WALTZZ on JavaScriptCore. However, in most runtimes, wasm-smith attains even lower edge coverage than AFL++, which substantiates the judgment in Section 3 that generative fuzzers may struggle to thoroughly traverse the input space. Secondly, even though RedQueen is generally considered superior to traditional fuzzers, its performance falls short when compared to AFL++. This can be attributed to the fact that while RedQueen can bypass certain roadblocks like magic values, it fails to account for the validity checks in Wasm, wasting a large fraction of time generating semi-valid inputs. A similar phenomenon has been observed with WasmFuzzer as well. Thirdly, the coverage improvement achieved by Wapplique over the initial seed corpus is negligible. This is unsurprising, as the capacity to generate diverse test cases for black-box fuzzers heavily relies on the initially collected code snippets. Moreover, Wapplique focuses mainly on four numeric types and is not able to fix certain Wasm contexts, which further constrains its performance.

Given that the primary objective of a fuzzer is to discover bugs, we present the number of bugs identified by each fuzzer in Table 2. All these bugs have been manually deduplicated using the stack traces provided by AddressSanitizer [47]. As shown, WALTZZ has detected the most bugs, outperforming its nearest competitor by 37.5%. No fuzzers find any bugs in JS engines, which is expected, since targets like V8 have been thoroughly tested by internal fuzzers. We will delve into the key attributes of historical bugs in the Wasm subsystem of JS engines in Section 7, which sheds light on why WALTZZ currently fails to detect new bugs in these engines. Moreover, as stated in the official wasmtime documentation, "*it is guaranteed that there is no undefined behavior or segfaults in either Wasm guest or the host itself,*" it is extremely difficult for existing fuzzers to identify memory corruption bugs or panics in Rust-based runtimes. Nevertheless, we have triaged and reported 20 previously unknown bugs found by WALTZZ, all of which have been confirmed, with 17 received CVE IDs. We further detail two critical bugs found by WALTZZ in Section 6.5. The first demands structural modifications to the code base, while the second presents a potential exploitation risk and is difficult to detect under normal conditions.

RQ1: WALTZZ surpasses existing state-of-the-art fuzzers by achieving 12.4% more code coverage and discovering 1.38× more bugs than the nearest competitor. WALTZZ has uncovered 20 new bugs in total which have all been verified and 17 CVE IDs are assigned.

6.3 Evaluation of Fuzzing Methods

To address RQ2 and evaluate the effectiveness of different mutation methods, we modify WALTZZ and add a counter for each method. These counters are incremented when the corresponding mutation methods generate an input that finds new code paths. We do not evaluate the efficacy of the generator as it is called on demand. We have conducted the experiment on three web embeddings over a 24-hour period and logged the counter values at one-minute intervals, which is consistent with the common practice observed in prior research [3, 22].

Figure 6 presents the relative contribution of each method over 24 hours averaged across 10 runs. Due to the diminishing discovery of new paths after the initial stage, we organize the data into differently-sized bins: 1-minute bins for the first hour, 5-minute bins up to the three-hour mark, 10-minute bins until the six-hour mark, and 20-minute bins to the end. Our analysis reveals that all mutators effectively contribute to uncovering new paths, with none displaying a disproportionate advantage. Furthermore, the effectiveness of mutators varies with different targets. As an example, the splicing mutator is particularly effective at finding new paths in SpiderMonkey, whereas it performs moderately on other targets.

RQ2: All mutation methods contribute to the effectiveness of WALTZZ and different targets potentially favor different mutators.

6.4 Validity of Generated Test Cases

To tackle RQ3, we conduct a 24-hour evaluation of WALTZZ on three web embeddings, during which we record the number of both valid and invalid inputs produced by our mutators and the generator. Considering that the generation of test cases is independent of the target program being tested, we believe the validity evaluation results can be confidently applied to the

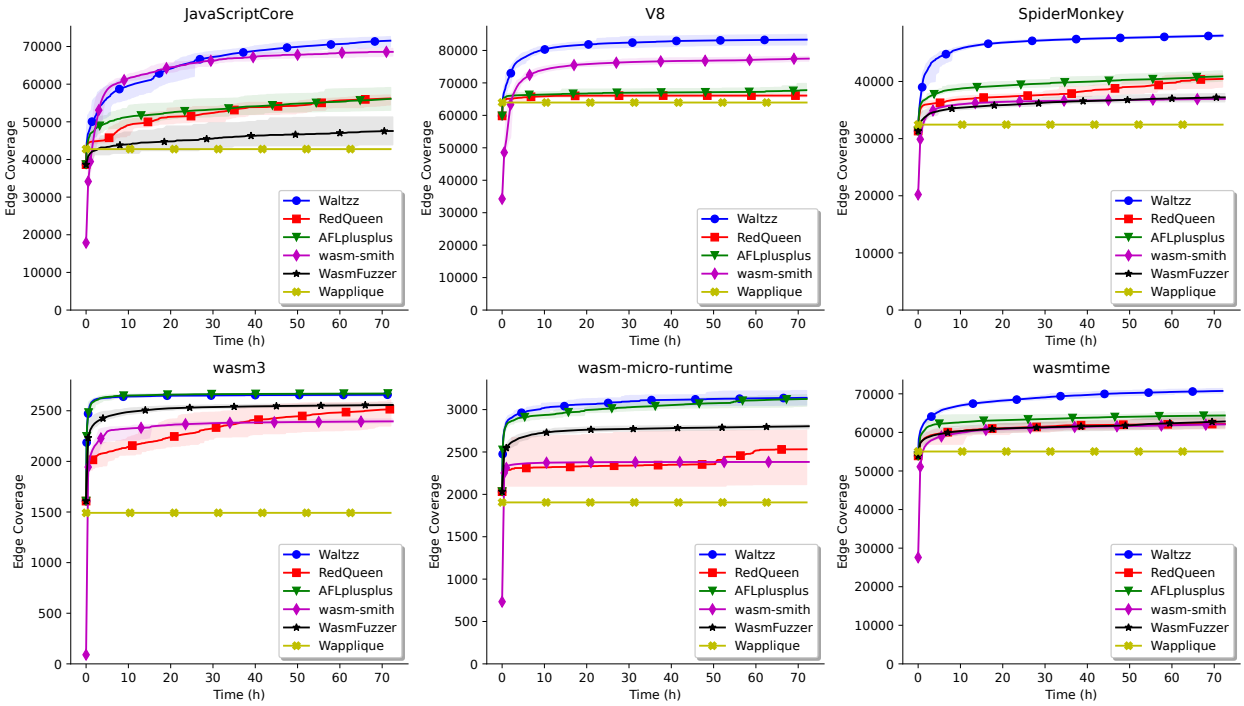


Figure 5: The edge coverage for all the evaluated techniques over time. The main line displays the mean across all runs, whereas the shaded area denotes the confidence band ranging from minimum value to maximum value.

non-web embeddings. Besides, we use wasm-validate from the Wasm binary toolkit [50] to ascertain the validity of these test cases.

Table 3 displays the validity rates. WALTZZ is able to generate hundreds of thousands of valid inputs with a success rate of 100%, a result enforced by the stack-invariant transformation. When compared to the data in Table 1, it is evident that WALTZZ produces a significantly higher number of valid inputs than AFL++. This capability enables WALTZZ to explore deeper states of the targets, which is a contributing factor to its superior performance relative to AFL++.

RQ3: WALTZZ can generate 100% valid inputs by virtue of the stack-invariant transformation, which empowers it to reach deeper states.

6.5 Case Studies

To answer RQ4, we present case studies of two bugs uniquely found by WALTZZ.

CVE-2024-33480. This bug is uncovered in wasm3, which triggers a stack overflow. wasm3 executes Wasm instructions through a sequence of operation calls and most operations are executed on an internal virtual stack. Furthermore, wasm3 is engineered with tail-call optimization, enabling 90% of the opcodes to operate without consuming the system stack.

To induce a stack overflow bug, the initial approach often involves constructing a simple recursive call that depletes the system stack. However, this method is impractical with wasm3, as it checks whether the maximum virtual stack capacity is reached at the function entrance, if so, wasm3 raises a `trapStackOverflow` error and aborts the execution. Besides, even in the absence of this safeguard, devising instructions that aggressively consume the system stack is challenging, as most opcodes do not interact with it.

However, a test case constructed by WALTZZ successfully bypasses the preceding sanity check and exhausts the entire stack allocated for wasm3. Figure 7 depicts a truncated PoC of this bug. Notably, the PoC includes a recursive call deeply nested within the loop structures. It appears that the `loop` instruction is not optimized for tail-call operations, as wasm3 must retain certain status information. Thus, each invocation

Table 3: The average counts of total and valid test cases from fuzzing three major JS engines across 10 runs, each with a 24-hour budget using WALTZZ.

Project	$Num_{Testcase}$	Num_{Valid}	$\frac{Num_{Valid}}{Num_{Testcase}}$
V8	452171.4	452171.4	100.0%
JavaScriptCore	394297.8	394297.8	100.0%
SpiderMonkey	527724.5	527724.5	100.0%
Avg.	458064.6	458064.6	100.0%

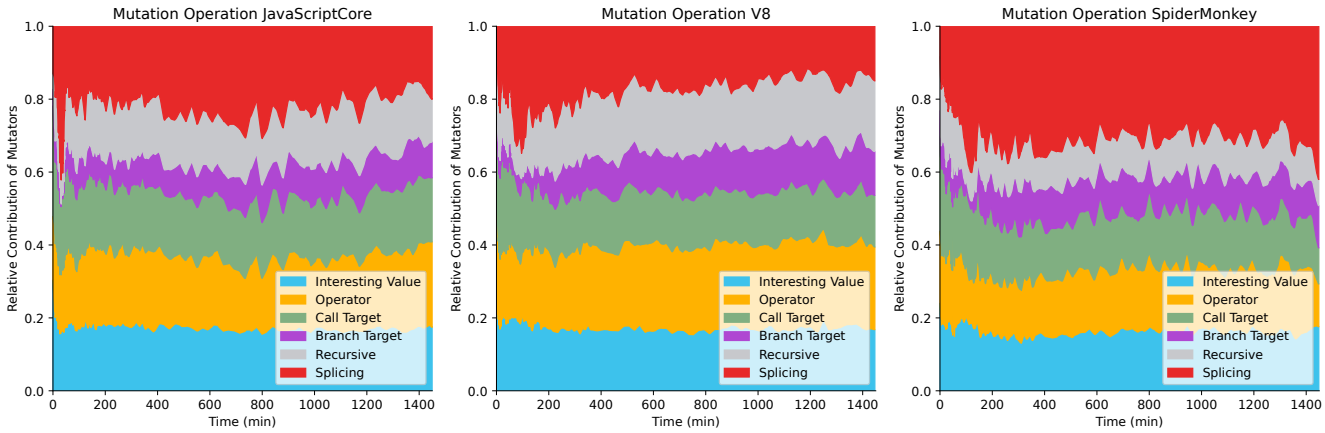


Figure 6: Percentage of identified new paths for each mutation method, averaged across 10 runs.

of the `op_Loop` consumes approximately `0x20` bytes of system stack space under the current setting. At the same time, the virtual stack only increases incrementally at the innermost call. This disparity ultimately leads to a stack overflow before the built-in sanity check could intervene. This bug relates to the tail-call architecture of wasm3, thus demanding structural changes to the code base.

```
(module
  (func (;0;)
    loop (result i32)
      loop (result i32)
        loop (result i32)
          loop (result i32)
            ;; several more loop (result i32)...
              call 0
              i32.const 150
            ;; several more end...
          end
        end
      end
    end
    drop)
  (func ...)
  (export "main" (func 0))
)
```

Figure 7: A truncated proof-of-concept code of CVE-2024-33480.

We argue that other baselines, such as AFL++ or RedQueen, are unlikely to discover this bug even with considerable time investment, as they struggle to generate valid test cases and probe deeper execution logic. Although wasm-smith might produce inputs that include recursive calls, it rarely generates instruction sequences that exhaust the system stack to trigger a stack overflow. In contrast, WALTZZ effectively constructs the PoC by initially employing a splicing mutator to combine two seed inputs, thereby crafting the necessary control structure. It then alters the target function of the `call` instruction to the function itself using the call target mutator. This underscores the synergy effect of our mutators, which helps WALTZZ to uncover bugs that elude other baselines.

```
(module
  (func $vul (result i32) (local f64)
    ... ;; wrong size check at the function entry
    block
      ...
      if
        f64.const 0x1.e848p+24
        ...
        f64.copysign ;; out-of-bounds write!
        call $long_param
        ...
      else
        ...
      end
      ...
    end)
  (func $main
    ... ;; code that almost saturates the virtual stack
    call $vul
    drop)
  (func $long_param (param f64 f64 f64 f64 f64 f64
                           f64 f64)
                    (result f64)
    ...)
  ...
  (export "main" (func $main)))
```

Figure 8: A truncated proof-of-concept code of CVE-2024-33477.

CVE-2024-33477. This vulnerability is also identified in wasm3, leading to an out-of-bounds write that corrupts the heap metadata. As previously mentioned, wasm3 utilizes a virtual stack, which is allocated on the heap with a predefined maximum size. Before execution, wasm3 will "compile" the function to calculate the amount of virtual stack space it will require. At the function entrance, wasm3 guarantees that the execution of the current function does not overflow the virtual stack. Therefore, Wasm inputs are normally prevented from writing outside the virtual stack.

Despite this, WALTZZ effectively generates a Wasm input taking advantage of a miscalculation bug in wasm3 to achieve

out-of-bounds write on the virtual stack. Figure 8 illustrates a truncated PoC of this bug. The PoC starts with a bunch of irrelevant operations designed solely to saturate the virtual stack. Upon invoking the function `$vul`, wasm3 attempts a "pre-compilation" to ascertain the number of required stack slots. However, it underestimates the slots required for the calling arguments of `$long_param`, mistakenly assuming that there has enough space on the virtual stack for `$vul`, thereby failing to trigger a `trapStackOverflow` error. This oversight permits the out-of-bounds write at `f64.copysign`. This bug could potentially be exploited for remote code execution by carefully structuring the code.

As per the developer, this vulnerability can be difficult to detect under normal circumstances. Fuzzers like wasm-smith often struggle to find this bug due to their inability to balance the avoidance of internal stack overflow errors with the out-of-bounds write. In contrast, WALTZZ leverages the skeleton-based generator to produce `$long_param` whose signature is crucial for triggering the vulnerability. This highlights the usefulness of our generator in generating rare code constructs.

RQ4: WALTZZ has identified vulnerabilities that are likely exploitable or necessitate structural modifications to the code base, demonstrating the practical impact of WALTZZ.

7 Discussion

Despite the efficacy of WALTZZ, there is still potential for further enhancement. In this section, we analyze a bunch of bugs from the SpiderMonkey Bugzilla database, presenting observations that could potentially strengthen the bug-finding capacity of WALTZZ.

Specifically, we curate a dataset from the SpiderMonkey Bugzilla. We consider exclusively memory failures, including crashes, assertion failures, and ASAN errors. Among these, we then select only those bugs that are consistently reproducible and accompanied by the PoC. This yields a dataset comprising exactly 100 memory bugs, which would be made available alongside the source code.

In the course of investigating each bug, we first concentrate on the aspects of build configuration and run command. The first observation is that different bugs typically require distinct build or run commands, and no single command is universally effective. Notably, nine bugs are identified under the ARM architecture, highlighting the potential benefits of targeting diverse architectures in finding bugs. Moreover, a large fraction of run commands adjust JIT tiers, e.g., by mandating test cases to be compiled by baseline compilers. Lastly, an interesting bug `Bug1500231` highlights the importance of experimenting with varied combinations of command-line arguments, which is mainly triggered by the `-enable-avx` option. In summary, diversifying target settings could empower WALTZZ to find more bugs, and we leave this as a future direction.

Wasm is an evolving language, with many proposals currently under standardization. Presently, WALTZZ supports only the Wasm core specification as this standard is the most widely adopted among runtimes. However, we are keen to assess the prevalence of bugs introduced by the implementation of ongoing proposals. Moreover, we also analyze whether interactions between Wasm modules and their host environments impact bug manifestation. Figure 9 shows the result: around 40% of the bugs involve proposals that are still in progress. Upon closer examination, we find that newer bugs tend to relate more frequently to recent proposals and the implementation of the core specification is less likely to have bugs nowadays. This agrees with the observation from AFLChurn [68] that most bugs have been introduced by recent code changes. Therefore, incorporating new proposals into WALTZZ should help it identify more new bugs and we consider this as future work. Additionally, 50% of the bugs interact with the JS contexts, e.g., bug `Bug1496362` occurs only when the tracelogger is activated with the Wasm input being simply `(module)`. This points us towards a viable direction of exploring the interplay between Wasm inputs and the host environment, particularly within the context of Wasm System Interface [51].

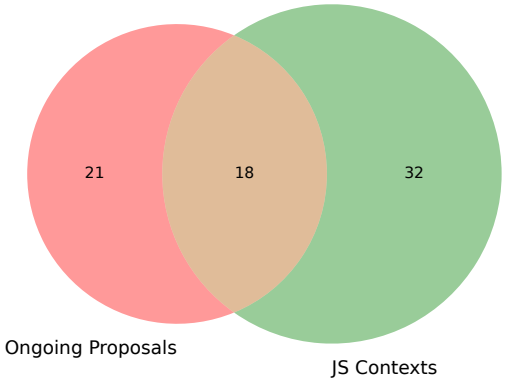


Figure 9: Number of bugs introduced by using ongoing proposals or interacting with JS contexts.

8 Related Work

8.1 Fuzzing

Greybox fuzzers are currently the mainstream in academia and industry, which utilize instrumentation feedback such as edge coverage to facilitate the exploration of the target input space. Since the advent of AFL [20], a large body of research work has tried to improve the efficiency of greybox fuzzing.

One trend pays more attention to improving the efficiency of different components in the greybox fuzzers. CollAFL [19] addresses the hash collision problem of the bitmap to achieve precise feedback. AFLFast [7], EcoFuzz [63], MobFuzz [65]

and other works [36, 48, 55] optimize which seed is selected first for mutation and how many test cases should be generated for each seed according to predefined criteria. MOpt [34] and DARWIN [26] utilize novel optimization algorithms for scheduling the mutation process, which achieve satisfactory results. Another line of work tends to infer more knowledge about the program under test. RedQueen [4], VUzzer [42] and other works [18, 33] try to conquer complex path constraints in a generic manner by tracking the data flow between input bytes and the operands in comparison instructions. However, target programs often accept inputs in a specific format which could be hard to satisfy by previous fuzzers. Thus, structure-aware fuzzers [3, 6, 15, 40, 56] synthesize test inputs by either providing a predefined specification or inferring the structure dynamically.

Inferring input structures is not enough, since inputs often carry semantics that are checked by the target. This is where fuzzers transition from general to specialized. Many works have been proposed to target different research areas such as kernel [8, 61], compiler [14, 62]. Since Wasm originates from the web scenario, we discuss some of the works targeting the JS engine. CodeAlchemist [24] fragments seed inputs into code bricks and annotates each brick with a set of constraints such as the type of a variable, and these bricks are later combined with all the constraints satisfied. Fuzzilli [22] designs a specific IR that focuses on discovering JIT compiler vulnerabilities in the JS engine. DIE [38] proposes aspect-preserving mutations that can maintain interesting properties in the seed inputs. While all of these works take advantage of the type system to ensure the semantic correctness of generated inputs and inspire our work to a large extent, we argue that different type systems demand entirely distinct designs. Maintenance of proper stack semantics presents a unique challenge in the Wasm context and cannot be addressed by previous works. This underscores the importance of WALTZZ.

8.2 WebAssembly Security

Wasm security can be roughly categorized into application security and host security. The former generally refers to the security implications introduced by Wasm binaries. The latter refers to the security issues in the Wasm implementations.

As for application security, the early adopters of Wasm have been the websites that use the computing resources of visitors to mine cryptocurrencies [30]. To mitigate this, several works [28, 57] have been proposed to detect and prevent malicious crypto mining. One technical paper [32] demonstrates that vulnerabilities in memory unsafe languages can transfer to the Wasm binaries and be exploited more easily due to the lack of common mitigation. Wasm could also be utilized as a part of the malware, Wobfuscator [44] moves parts of the computation from JS to Wasm and evades lots of malware detectors. JWBinder [60] proposes an inter-language program dependency graph to detect multilingual malware of this type.

For host security, Google Project Zero demonstrates an early study [41] towards the vulnerabilities in the Wasm implementations. WasmFuzzer [27] applies simple mutations on the Wasm modules such as adding a global variable. WARF [17] tries out different fuzzing techniques such as structure-aware fuzzing to find bugs in Wasm implementations. While WARF incorporates various advanced fuzzing techniques, it demands manually crafted Rust test harnesses, and its support for many existing Wasm runtimes remains limited. Wasm-smith [10] monitors the stack states during the test case generation, i.e., the instruction generated by wasm-smith depends on current stack types. While enforcing the semantic correctness, some instructions can hardly be generated by wasm-smith since the required stack types are difficult to meet, thereby hindering the diversity of its generated test cases. WADIFF [67] leverages symbolic execution to generate inputs for each instruction and conducts differential testing. However, WADIFF can generate simple Wasm modules only and the use of symbolic execution is heavyweight. WASMaker [13] and Wapplique [66] are two concurrent works that guarantee the validity of generated test cases. WASMaker generates Wasm modules by disassembling and assembling existing Wasm modules, a process similar to that used by CodeAlchemist [24]. Wapplique utilizes a single mutator akin to splicing to produce a predetermined number of new inputs. Both of them are black-box fuzzers and their capacity to generate diverse test cases largely depends on the initially collected seed corpus. Unlike these works, WALTZZ proposes the concept of stack-invariant transformation and integrates it into the design of mutators and the generator to effectively explore the target runtimes at a minimal cost.

9 Conclusion

In this paper, we introduce WALTZZ, an effective greybox fuzzing framework tailored for Wasm runtimes. We propose a novel approach termed stack-invariant transformation, which effectively maintains the desired stack semantics and serves as a foundational principle throughout the design of WALTZZ. We then devise a suite of Wasm mutators that collaboratively explore different combinations of the control and data flow. Besides, we design a skeleton-based generation algorithm, facilitating the creation of rare code constructs. We have implemented WALTZZ and evaluated it on seven Wasm runtimes. Compared to the leading fuzzers, WALTZZ distinguishes itself by achieving 12.4% more code coverage and detecting 1.38× more unique bugs than its nearest competitor. In total, WALTZZ has identified 20 new bugs which have all been verified and 17 CVE IDs have been assigned.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their insightful comments on our work. This research is supported

in part by the Keysight research award, the Cisco research award, the Key research and development project of Zhejiang Province (No.2024C01G2013895), the Open Research Fund of Engineering Research Center of Blockchain Application, Supervision And Management (Southeast University), Ministry of Education, the NSFC under Grant No. 62302443, and the Fellowship of China National Postdoctoral Program for Innovative Talents (BX20230307).

Ethics Considerations

We have responsibly reported all identified bugs to the relevant vendors. The two memory bugs discussed in Section 6.5 have been confirmed by the developers and appropriate mitigation strategies have been formulated. To thwart potential exploitation by attackers, we have intentionally truncated the PoC, omitting most details. We believe these measures adequately address the ethical concerns.

Open Science

To comply with the open science policy, we have made the prototype of WALTZZ and the memory bug dataset introduced in Section 7 available at https://zenodo.org/records/14718828.

References

[1] AFL++. The custom mutator interface in AFL++. https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/custom_mutators.md. Accessed Jun. 2024.

[2] Apple. JavaScriptCore, built-in JavaScript engine for WebKit. https://docs.webkit.org/Deep%20Dive/JSC/JavaScriptCore.html. Accessed Jun. 2024.

[3] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: fishing for deep bugs with grammars. In *26th Annual Network and Distributed System Security Symposium*, 2019.

[4] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: fuzzing with input-to-state correspondence. In *26th Annual Network and Distributed System Security Symposium*, 2019.

[5] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. Jit-picking: Differential fuzzing of javascript engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 351–364, 2022.

[6] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: synthesizing structure while fuzzing. In *28th USENIX Security Symposium*, pages 1985–2002, 2019.

[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[8] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. No grammar, no problem: Towards fuzzing the linux kernel without system-call descriptions. In *30th Annual Network and Distributed System Security Symposium*, 2023.

[9] Bytecode Alliance. wasm-micro-runtime, lightweight standalone Wasm runtime. https://github.com/bytecodealliance/wasm-micro-runtime. Accessed Jun. 2024.

[10] Bytecode Alliance. wasm-smith, a Wasm test case generator. https://github.com/bytecodealliance/wasm-tools/tree/main/crates/wasm-smith. Accessed Jun. 2024.

[11] Bytecode Alliance. wasmtime, a fast and secure runtime for WebAssembly. https://github.com/bytecodealliance/wasmtime. Accessed Dec. 2024.

[12] CanSecWest. The Pwn2Own hacking contest. https://www.secwest.net/pwn2own. Accessed Jun. 2024.

[13] Shangtong Cao, Ningyu He, Xinyu She, Yixuan Zhang, Mu Zhang, and Haoyu Wang. Wasmaker: Differential testing of webassembly runtimes via semantic-aware binary generation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1262–1273, 2024.

[14] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. Grayc: Greybox fuzzing of compilers and analysers for C. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1219–1231, 2023.

[15] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. WEIZZ: automatic grey-box fuzzing for structured binary formats. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1–13, 2020.

[16] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies*, 2020.

[17] FuzzingLabs. The WebAssembly runtimes fuzzing project. https://github.com/FuzzingLabs/wasm_runtimes_fuzzing. Accessed Jun. 2024.

[18] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: data flow sensitive fuzzing. In *29th USENIX Security Symposium*, pages 2577–2594, 2020.

[19] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy*, pages 679–696, 2018.

[20] Google. American fuzzy lop. https://github.com/google/AFL. Accessed Jun. 2024.

[21] Google. V8, open source high-performance JavaScript and WebAssembly engine. https://v8.dev/. Accessed Jun. 2024.

[22] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. FUZZILLI: fuzzing for javascript JIT compiler vulnerabilities. In *30th Annual Network and Distributed System Security Symposium*, 2023.

[23] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[24] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *26th Annual Network and Distributed System Security Symposium*, 2019.

[25] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the 21th USENIX Security Symposium*, pages 445–458, 2012.

[26] Patrick Jauernig, Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf, and Ahmad-Reza Sadeghi. DARWIN: survival of the fittest fuzzing mutators. In *30th Annual Network and Distributed System Security Symposium*, 2023.

[27] Bo Jiang, Zichao Li, Yuhe Huang, Zhenyu Zhang, and W. K. Chan. Wasmfuzzer: A fuzzer for wasassembly virtual machines. In *The 34th International Conference on Software Engineering and Knowledge Engineering*, pages 537–542, 2022.

[28] Amin Kharraz, Zane Ma, Paul Murley, Charles Lever, Joshua Mason, Andrew Miller, Nikita Borisov, Manos Antonakakis, and Michael D. Bailey. Outguard: Detecting in-browser covert cryptocurrency mining in the wild. In *The World Wide Web Conference*, pages 840–852, 2019.

[29] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[30] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. 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*, pages 1714–1730, 2018.

[31] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. Montage: A neural network language model-guided javascript engine fuzzer. In *29th USENIX Security Symposium*, pages 2613–2630, 2020.

[32] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of webassembly. In *29th USENIX Security Symposium*, pages 217–234, 2020.

[33] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. PATA: fuzzing with path aware taint analysis. In *43rd IEEE Symposium on Security and Privacy*, pages 1–17, 2022.

[34] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium*, pages 1949–1966, 2019.

[35] Chenyang Lyu, Shouling Ji, Xuhong Zhang, Hong Liang, Binbin Zhao, Kangjie Lu, and Raheem Beyah. EMS: history-driven mutation for coverage-based fuzzing. In *29th Annual Network and Distributed System Security Symposium*, 2022.

[36] Chenyang Lyu, Hong Liang, Shouling Ji, Xuhong Zhang, Binbin Zhao, Meng Han, Yun Li, Zhe Wang, Wenhai Wang, and Raheem Beyah. SLIME: program-sensitive energy allocation for fuzzing. In *31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 365–377, 2022.

[37] Mozilla. SpiderMonkey, Mozilla's JavaScript and WebAssembly engine. https://spidermonkey.dev/. Accessed Jun. 2024.

[38] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy*, pages 1629–1642, 2020.

[39] Manfred Paul. The exploitation for a type confusion bug in Google V8's WebAssembly subsystem. https://www.zerodayinitiative.com/blog/2024/5/2/cve-2024-2887-a-pwn2own-winning-bug-in-google-chrome. Accessed Jun. 2024.

[40] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Razvan Caciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Trans. Software Eng.*, 47(9):1980–1997, 2021.

[41] Project Zero. The problems and promise of WebAssembly. https://googleprojectzero.blogspot.com/2018/08/the-problems-and-promise-of-webassembly.html. Accessed Jun. 2024.

[42] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *24th Annual Network and Distributed System Security Symposium*, 2017.

[43] RET2Systems. The exploitation for an integer overflow bug in Apple JavaScriptCore's Wasm subsystem. https://blog.ret2.io/2021/06/02/pwn2own-2021-jsc-exploit/. Accessed Jun. 2024.

[44] Alan Romano, Daniel Lehmann, Michael Pradel, and Weihang Wang. Wobfuscator: Obfuscating javascript malware via opportunistic translation to webassembly. In *43rd IEEE Symposium on Security and Privacy*, pages 1574–1589, 2022.

[45] Rust Fuzzing Authority. afl.rs, fuzzing Rust code with AFLplusplus. https://github.com/rust-fuzz/afl.rs. Accessed Dec. 2024.

[46] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. Token-level fuzzing. In *30th USENIX Security Symposium*, pages 2795–2809, 2021.

[47] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference*, pages 309–318, 2012.

[48] Dongdong She, Abhishek Shah, and Suman Jana. Effective seed scheduling for fuzzing with graph centrality analysis. In *43rd IEEE Symposium on Security and Privacy*, pages 2194–2211, 2022.

[49] The WebAssembly Community Group. Binaryen, a compiler and toolchain infrastructure library for Wasm. https://github.com/WebAssembly/binaryen. Accessed Jun. 2024.

[50] The WebAssembly Community Group. WABT, the WebAssembly binary toolkit. https://github.com/WebAssembly/wabt. Accessed Jun. 2024.

[51] The WebAssembly Community Group. WASI, WebAssembly system interface. https://github.com/WebAssembly/WASI. Accessed Jun. 2024.

[52] W3C. The official tests for the core WebAssembly semantics. https://github.com/WebAssembly/spec/tree/main/test/core. Accessed Jun. 2024.

[53] W3C. The WebAssembly core specification. https://webassembly.github.io/spec/core/. Accessed Jun. 2024.

[54] W3C. The WebAssembly-JavaScript API. https://webassembly.github.io/spec/js-api/. Accessed Jun. 2024.

[55] Jinghan Wang, Chengyu Song, and Heng Yin. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In *28th Annual Network and Distributed System Security Symposium*, 2021.

[56] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering*, pages 724–735, 2019.

[57] Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W. Hamlen, and Shuang Hao. SEISMIC: secure in-lined script monitors for interrupting cryptojacks. In *23rd European Symposium on Research in Computer Security*, volume 11099 of *Lecture Notes in Computer Science*, pages 122–142, 2018.

[58] wasm3. wasm3, a fast WebAssembly interpreter and the most universal Wasm runtime. https://github.com/wasm3/wasm3. Accessed Jun. 2024.

[59] wasmCloud. wasmCloud, an open source CNCF project that enables deployment of polyglot Wasm apps across any cloud. https://wasmcloud.com/. Access Sept. 2024.

[60] Yifan Xia, Ping He, Xuhong Zhang, Peiyu Liu, Shouling Ji, and Wenhai Wang. Static semantics reconstruction for enhancing javascript-webassembly multilingual malware detection. In *28th European Symposium on Research in Computer Security*, volume 14345 of *Lecture Notes in Computer Science*, pages 255–276, 2023.

[61] Jiacheng Xu, Xuhong Zhang, Shouling Ji, Yuan Tian, Binbin Zhao, Qinying Wang, Peng Cheng, and Jiming Chen. Mock: Optimizing kernel fuzzing mutation with context-aware dependency. In *31st Annual Network and Distributed System Security Symposium*, 2024.

[62] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–294, 2011.

[63] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In *29th USENIX Security Symposium*, pages 2307–2324, 2020.

[64] Alon Zakai. Emscripten: an llvm-to-javascript compiler. In *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 301–312, 2011.

[65] Gen Zhang, Pengfei Wang, Tai Yue, Xiangdong Kong, Shan Huang, Xu Zhou, and Kai Lu. Mobfuzz: Adaptive multi-objective optimization in gray-box fuzzing. In *29th Annual Network and Distributed System Security Symposium*, 2022.

[66] Wenxuan Zhao, Ruiying Zeng, and Yangfan Zhou. Wapplique: Testing webassembly runtime via execution context-aware bytecode mutation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1035–1047, 2024.

[67] Shiyao Zhou, Muhui Jiang, Weimin Chen, Hao Zhou, Haoyu Wang, and Xiapu Luo. WADIFF: A differential testing framework for webassembly runtimes. In *38th IEEE/ACM International Conference on Automated Software Engineering*, pages 939–950, 2023.

[68] Xiaogang Zhu and Marcel Böhme. Regression greybox fuzzing. In *2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2169–2182, 2021.

Appendix

```rust
#[macro_use]
extern crate afl;
extern crate wasmtime;

fn main() {
    fuzz!(|data: &[u8]| {
        let engine = wasmtime::Engine::default();
        let mut store = wasmtime::Store::new(&engine, ());
        let Ok(module) = wasmtime::Module::from_binary(&engine, data) else { return };
        let Ok(instance) = wasmtime::Instance::new(&mut store, &module, &[]) else { return };
        let Ok(main_func) = instance.get_typed_func::<(), ()>(&mut store, "main") else { return };
        let _ = main_func.call(&mut store, ());
    });
}
```

Figure 10: Test harness for wasmtime.

Table 4: Summary of vulnerabilities found by WALTZZ. In the **Status** column, **Confirmed** indicates that the bug has been verified by developers, **Patched** signifies that an appropriate code patch has been proposed but not yet merged, **Fixed** denotes that the bug has been fixed in the main branch.

Target	CVE/Issue-ID	Bug Type	Crash Location (Function)	Status
wasm3	CVE-2024-33473	OOB-Write	`op_CopySlot_64`	Fixed
	CVE-2024-33474	OOB-Read	`op_Select_i64_ssr`	Confirmed
	CVE-2024-33475	OOB-Read	`RemoveCodePageOfCapacity`	Confirmed
	CVE-2024-33476	OOB-Write	`op_SetSlot_i64`	Confirmed
	CVE-2024-33478	OOB-Read	`op_Select_i64_srs`	Confirmed
	CVE-2024-33479	OOB-Read	`op_Select_f32_ssr`	Confirmed
	CVE-2024-33480	Stack Overflow	`op_Entry`	Confirmed
	CVE-2024-33481	OOB-Read	`op_Select_f64_rsr`	Confirmed
	CVE-2024-33482	OOB-Read	`op_MemCopy`	Confirmed
	CVE-2024-34247	OOB-Read	`op_Select_f32_rsr`	Confirmed
	CVE-2024-34248	OOB-Read	`Environment_ReleaseCodePages`	Confirmed
	CVE-2024-33477	Heap Buffer Overflow	`op_SetSlot_f64`	Patched
	CVE-2024-34246	OOB-Read	`main`	Patched
	CVE-2024-34249	Heap Buffer Overflow	`DeallocateSlot`	Patched
	CVE-2024-34252	Global Buffer Overflow	`PreserveRegisterIfOccupied`	Patched
wasm-micro-runtime	CVE-2024-34250	Heap Buffer Overflow	`wasm_loader_check_br`	Fixed
	CVE-2024-34251	OOB-Read	`block_type_get_arity`	Fixed
wasm-interp	Issue2310	OOB-Read	`wabt::interp::DataSegment::IsValidRange`	Fixed
	Issue2311	OOB-Write	`wabt::interp::DataSegment::Drop`	Fixed
	Issue2398	OOB-Read	`wabt::interp::FreeList<wabt::interp::Object*>::IsUsed`	Fixed

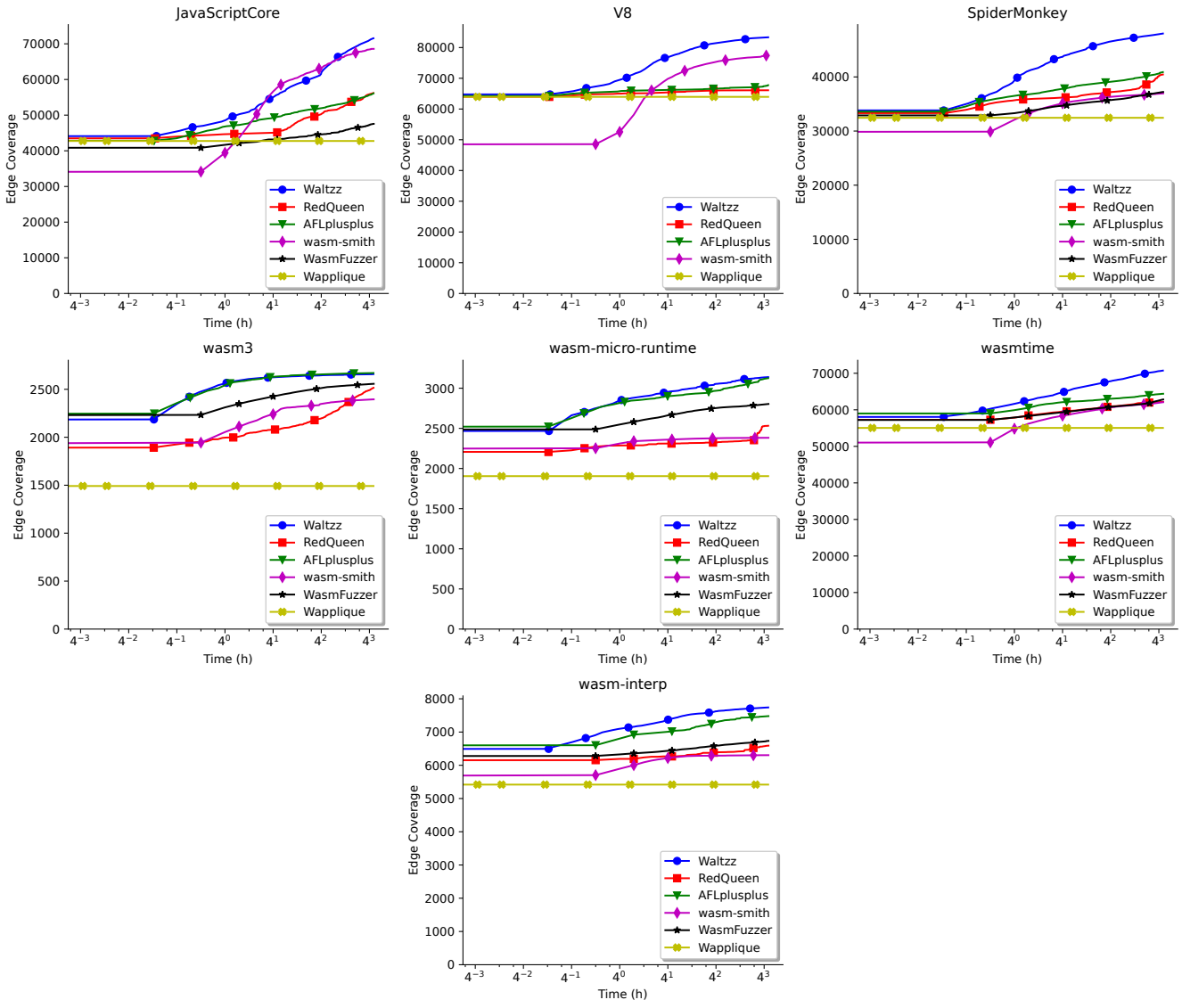


Figure 11: The edge coverage for all the evaluated techniques over a logarithmic time scale. The main line displays the mean across all runs, whereas the shaded area denotes the confidence band ranging from minimum value to maximum value.