Fuzzing JavaScript Engines by Fusion of JavaScript and WebAssembly Code

Anonymous Author(s)

ABSTRACT

JavaScript engines are a fundamental part of modern browsers, and many efforts have been invested in testing them to enhance their security. However, the incorporation of WebAssembly into JavaScript engines introduces new attack surfaces that have not received sufficient attention. Existing fuzzers for JavaScript engines primarily focus on JavaScript, neglecting WebAssembly code and its interactions with JavaScript. We introduce Mad-Eye, the first fuzzer that can test the JavaScript-WebAssembly interaction by fusing the JavaScript and WebAssembly code to uncover vulnerabilities in JavaScript engines. Mad-Eye employs a probing strategy to automate JavaScript-WebAssembly interactions and integrates WebAssembly regression tests to enhance fuzzing effectiveness. Evaluations of Mad-Eye on V8, SpiderMonkey, and JavaScriptCore detected 20 previously unknown vulnerabilities, with 17 confirmed and 11 fixed and merged into mainstream browsers, who acknowledged our reports with vulnerability bounties. 18 of the discovered vulnerabilities involve WebAssembly features, emphasizing the critical need to test WebAssembly executions within JavaScript engines. The tool is open-source and available to support further research.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

JavaScript Engines; WebAssembly; Fuzzing

ACM Reference Format:

Anonymous Author(s). 2025. Fuzzing JavaScript Engines by Fusion of JavaScript and WebAssembly Code. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25), October 13-17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

JavaScript engines are vital to modern web applications, playing a crucial role in browser responsiveness and overall user experience. Given that these engines execute untrusted client-side code, ensuring their security has become a paramount concern for both researchers and industry professionals. Over the years, substantial efforts have been dedicated to testing the security of these engines. These studies have primarily focused on identifying vulnerabilities and bugs within JavaScript engines [33, 38–42, 44, 48, 51–53], typically by generating random JavaScript code and analyzing the resulting crashes or unexpected outputs.

In addition to JavaScript code, WebAssembly (Wasm) code has emerged as a critical input for JavaScript engines to facilitate high-performance applications, introducing new attack vectors that have yet to be fully addressed. The execution of malicious Wasm code can also create opportunities for exploitation, potentially leading to engine crashes [20] or, in severe cases, remote code execution (RCE) [5, 6]. These emerging threats underscore the necessity of incorporating this additional input vector into ongoing research to enhance the security of JavaScript engines.

```
1  // Compile a WebAssembly module
2  load("test/mjsunit/wasm/wasm-module-builder.js");
3  const builder = new WasmModuleBuilder();
4  const func = builder.addImport("m", "i", kSig_v_v);
5  builder.addFunction("main", kSig_v_v)
6         .addBody([kExprCallFunction,func])
7         .exportFunc();
8  const v27 = new WebAssembly.Suspending(() => { return
       Promise.resolve(); });
9  const instance = builder.instantiate({"m": {"i": v27
       }});
10
11 // JavaScript-WebAssembly interactions
12 function f() {
13     delete [][instance];
14 }
15 let p = WebAssembly.promising(instance.exports.main);
16 p();
17 instance.toString = f;
18 instance.toString(); // v8 crash here
```

Listing 1: A crash caused by JS-Wasm interactions detected by Mad-Eye in V8.

Recent works have explored generating diverse WebAssembly (Wasm) code for testing Wasm runtimes. For instance, Park *et al.* [47] proposed a reverse stack-based technique to generate random Wasm code. Zhao *et al.* [54] introduced an execution context-aware mutation approach to enhance Wasm code diversity. Similarly, Cao *et al.* [35] disassembled and reassembled real-world Wasm binaries to stress-test runtimes. In the industry, Google employs an internal fuzzing driver [2] that generates various Wasm opcodes to evaluate the V8 engine's compilation and execution processes. However, our analysis indicates that these approaches face several limitations in effectively testing JavaScript engines.

First, while previous works have focused on generating standalone WebAssembly or JavaScript code, it overlooks a critical aspect of JavaScript engine testing: the interactions between Wasm and JavaScript code. Effective testing of these interactions is crucial because Wasm runs in a sandbox within JavaScript engines, and any interaction bypassing this sandbox can have severe security implications. For example, an in-the-wild exploit revealed that a vulnerability (CVE-2023-2033) in V8's Wasm import feature could bypass the sandbox, enabling an arbitrary write primitive [5]. Such incidents are not one-offs. Listing 1 illustrates a minimized exploit for a new vulnerability detected by our tool in V8. In this case, lines 11–18 contain JavaScript code interacting with Wasm code in lines 1–9, leading to a crash in V8 at line 18. Similarly, Listing 3 demonstrates another newly discovered vulnerability where

JavaScript-Wasm interactions result in type confusion in V8.[1] Such vulnerabilities cannot be detected by existing approaches, which either generate Wasm or JavaScript code without considering their interactions. If exploited, these vulnerabilities could have severe consequences [19].

Second, previous approaches to generating Wasm code either create it randomly from scratch [2, 47, 54] or rely on real-world Wasm binaries [35]. In contrast, Wasm regression tests [23–25] are crafted from real-world exploits identified by security analysts [5, 20] or unit tests written by engine developers. These tests often involve both Wasm and JavaScript code influencing Wasm execution. Purely generating or mutating Wasm code tends to miss these nuanced relationships. To address this, while our tool can generate arbitrary Wasm and JavaScript code to discover new vulnerability patterns, we prioritize leveraging Wasm regression tests to effectively trigger vulnerabilities tied to previously identified risky functionalities. For instance, our tool uses a Wasm regression test [21] to produce Listing 1, which incorporates JavaScript `Promise` features—key to triggering the vulnerability. Similarly, Listing 4 showcases another new vulnerability, discovered by mutating a Wasm unit test [4], that Apple later assigned *CVE-2024-anonymous* with a high-severity rating. Other approaches fail to detect these vulnerabilities within the same fuzzing time budget.

This paper introduces Mad-Eye, the first fuzzer capable of utilizing Wasm regressions and generating both JavaScript and Wasm code in an *interdependent* manner to test JavaScript engines. The key to its success lies in the design incorporating the two aforementioned insights. To achieve this, we address two main challenges. The first challenge is enabling complex interactions between Wasm and JavaScript. Previous code generation approaches cannot produce interoperable code across different languages because the resources defined in one language are not inherently visible to the other. This challenge becomes even more complex when both JavaScript and Wasm code are randomly generated, causing their imported and exported resources (*e.g.*, WebAssembly function names exported to JavaScript) to evolve dynamically. To address this, we propose *wasm-generator*, the first generator capable of generating Wasm modules that support interaction from the Wasm side. Meanwhile, Mad-Eye's JavaScript-side generator produces diverse JavaScript code to define WebAssembly imports and utilize WebAssembly exports. To bridge the cross-language gap that causes unawareness of imported and exported resources, Mad-Eye employs a recursive *probing* strategy. By instrumenting code, Mad-Eye identifies and extracts the properties of objects exchanged between JavaScript and WebAssembly, enabling awareness of available imported and exported resources for automated interaction synthesis.

The second challenge is generating high-quality test cases when mutating both JavaScript and Wasm code. To address this, we propose distinct mutation strategies for each. For Wasm, we translate its code into WasmBuilder APIs [28], a JavaScript-based representation introduced by V8. This framework provides a unified way to represent both JavaScript and Wasm code. It facilitates mutations on test cases involving both languages and integrates seamlessly with Wasm regression tests [23–25], which JavaScript engine developers maintain in the WasmBuilder format. To generate new Wasm code, we propose a constrained mutation for WasmBuilder APIs that adheres to their structural and semantic constraints. On the JavaScript side, upon the probing strategy, we introduce an approach called *Variable-Guided Generation*, which prioritizes the use of Wasm resources in JavaScript code generation to increase interaction complexity. During JavaScript code mutation, upon existing mutators, we introduce an *Object-Shape Aware* method that leverages more fine-grained shape information [13] of WebAssembly objects for accurate WebAssembly object mutation.

We evaluate Mad-Eye on three major JavaScript engines: V8, SpiderMonkey, and JavaScriptCore. After manual deduplication, Mad-Eye found 20 previously unknown security vulnerabilities: seven in V8, four in SpiderMonkey, and nine in JavaScriptCore. As of this writing, browser developers have confirmed 17 vulnerabilities and fixed 11 of them. Patching several confirmed vulnerabilities requires collaboration with external developers and is pending. We also perform ablation studies to evaluate the contributions of Mad-Eye's two key components. The experiments show that leveraging regressions significantly improves Wasm code coverage, while enabling JavaScript-Wasm interactions uncovers new interaction vulnerabilities. To compare Mad-Eye with state-of-the-art tools, we evaluate existing works RGFuzz [47], Fuzzilli [39], WASMaker [35], and *wasm-generator* [2]. Mad-Eye demonstrates the best performance in detecting vulnerabilities and achieving higher coverage for code associated with Wasm functionality.

In summary, this work has the following contributions:

- We identify previously overlooked attack vectors imposed by the execution of Wasm code in JavaScript engines; we then present two insights for detecting vulnerabilities through the execution of JavaScript and Wasm code.
- We implemented these insights in a tool called Mad-Eye, which is open-source and available at https://anonymous.4open.science/r/Mad-Eye-1B2E.
- Mad-Eye detected 20 vulnerabilities across three major JavaScript engines; 17 have been confirmed, and 11 have been fixed by the browser developers.

2 BACKGROUND

2.1 WebAssembly, WebAssembly JavaScript APIs, and WasmBuilder APIs

In this subsection, we explain how Wasm executes in JavaScript engines, how it interacts with JavaScript environment, and its representation using WasmBuilder APIs.

2.1.1 Compilation and Execution of Wasm Code. The process of creating and executing Wasm code involves several stages:

- **Source Code to Wasm Binary:** High-level programming languages are compiled into Wasm modules using compilers such as `Emscripten` [10] for C/C++. These modules are structured as compact bytecode files (`.wasm`).
- **Compilation and Executon:** A `.wasm` module is executed using a WebAssembly runtime, which can either be standalone, such as `Wasmer` [29], or embedded into JavaScript engines. The

[1] Google rated this type confusion vulnerability as high-severity, assigned it CVE-2024-anonymous, and awarded us $8,000 for the discovery.

runtime first validates the module, then compiles its bytecode into native machine code and executes it. In JavaScript engines, Wasm modules are typically compiled ahead of time and then executed, interleaving with JIT optimizers (*e.g.*, Turbofan in V8).

2.1.2 WasmBuilder APIs. V8 introduces the *WasmBuilder* JavaScript library [28] to simplify the creation of Wasm modules. The WasmBuilder consists of various APIs to programmatically construct Wasm bytecode in JavaScript. In Listing 2, lines 2-11 is an example of constructing a Wasm module via WasmBuilder APIs. The equivalent WAT format of this WasmBuilder program is shown in lines 13-23. The program first loads the WasmBuilder library (line 2) and constructs a builder instance (line 3). From lines 4 to 11, it calls different WasmBuilder APIs to assemble a Wasm module. Specifically, each API emits corresponding Wasm bytecode to a buffer, which is then passed to `WebAssembly.Module()` in line 24 to be compiled. For example, `addImportedGlobal()` appends a declaration of a `Global` object with variable name `env::var` to the bytecode's import section; `addBody()` emits organized Wasm instructions (*e.g.*, `kExprLocalGet` is a predefined Wasm opcode inside `"wasm-module-builder.js"`) to the bytecode's function section.

WasmBuilder APIs facilitate convenient manipulation of Wasm module structures without the need to generate higher-level code (*e.g.*, C++/Rust) and compile it into Wasm bytecode. Since JavaScript engines can directly execute WasmBuilder to construct and execute Wasm modules, most Wasm regression tests are manually written by the JavaScript engine developers *in the format of WasmBuilder API calls* [23–25].

2.1.3 JavaScript-Wasm Interactions and JavaScript WebAssembly APIs. JavaScript-Wasm interactions work as follows: JavaScript can create objects to be imported into and used by Wasm modules (*imports*); and Wasm modules can also export Wasm objects (*exports*) into JavaScript code for access. There are five types of objects, including *Table* (used to store references to Wasm functions for dynamic dispatch), *Memory* (used for managing linear Wasm memory), *Global* (used as global variables in Wasm), *Tag* (used for structured exception handling) and regular Wasm/JavaScript *Function*. They can be defined within a Wasm module as *internal objects* and (optionally) exported to JavaScript, or can also be defined by JavaScript through JavaScript WebAssembly APIs [31] (*e.g.*, `new WebAssembly.Global()`) and imported into a Wasm module.

We use the code in Listing 2 as an example. In line 24, the Wasm bytecode, generated by `builder.toBuffer()` and comprising sections such as type, import, and function, is passed to the `WebAssembly.Module()` constructor for compilation. The compiled module is subsequently instantiated in line 28 using the `WebAssembly.Instance()` constructor, enabling its execution and interaction with JavaScript. Modules with imported objects must declare these dependencies in their import section. The JavaScript environment is responsible for defining these imports (line 26) and linking them during instantiation (lines 27–28). Additionally, imported objects can be directly manipulated within JavaScript, as demonstrated in line 29. Exported objects, declared in the module's export section, are made accessible in the JavaScript environment via the `exports` property of the instantiated `wasmInstance`. For example, line 30 demonstrates how an exported Wasm function is invoked from the JavaScript environment.

```
1  // WasmBuilder APIs
2  load('test/mjsunit/wasm/wasm-module-builder.js'); // a
       library provided by V8.
3  const builder = new WasmModuleBuilder();
4  builder.addImportedGlobal('env', 'var', kWasmI32);
5  builder.addFunction('add', kSig_i_i)
6    .addBody([
7      kExprLocalGet, 0,
8      kExprGlobalGet, 0,
9      kExprI32Add
10   ])
11   .exportFunc();
12 /* its equivalent WAT format
13 (module
14   (type (func (param i32) (result i32)))
15   (import "env" "var" (global i32)) ;; import section
16   (func $add (type 0) (param i32) (result i32)
17     local.get 0       ;; Load the first parameter
18     global.get 0      ;; Use imports
19     i32.add           ;; Add them
20   )
21   (export "add" (func $add)) ;; export section
22 )
23 */
24 let module = WebAssembly.Module(builder.toBuffer());
25 // JavaScript interactions
26 let glob = new WebAssembly.Global({value: 'i32',
       mutable: true});
27 let imports = {env: {var: glob}};
28 const wasmInstance = new WebAssembly.Instance(module,
       imports);
29 glob.value = 10;
30 wasmInstance.exports.add(3);  // Use exports
```

Listing 2: An example of a WasmBuilder program, its equivalent WAT format, and its interaction with JavaScript code.

2.2 Protection Mechanisms of Wasm Executions in JavaScript Engines

Wasm execution in JavaScript engines incorporates several protection mechanisms to secure the execution of untrusted Wasm code. These mechanisms isolate Wasm from the host environment and implement defenses, such as control flow integrity [17], to protect against attacks targeting JavaScript engines.

Sandboxing. Wasm code is isolated from the host system using a sandbox [26]. This ensures that even if the Wasm code is corrupted or crashes, it cannot compromise the integrity of the JavaScript engine. The isolation is enforced by running Wasm in a separate memory space. This also prevents Wasm from affecting the performance or stability of JavaScript execution.

Additionally, Wasm code cannot directly access JavaScript's runtime APIs, such as the DOM or other sensitive browser features. All interactions between Wasm and JavaScript are strictly mediated through the imports and exports. They ensure that only explicitly imported JavaScript code is executed in Wasm, and vice versa.

Securing WebAssembly Execution. JavaScript engines monitor Wasm execution to detect and prevent malicious or unexpected behavior. This includes classic protections such as validating memory accesses, checking function call integrity, and enforcing execution boundaries [27]. Besides protecting the host environment, these

checks mitigate attacks targeting the Wasm environment (*e.g.*, exploitation of vulnerable Wasm modules).

2.3 Threat Model and Existing Works

In this subsection, we present our threat model and existing works on Wasm and JavaScript runtime security.

2.3.1 Threat Model. We focus on generating a mix of JavaScript and Wasm code to trigger vulnerabilities in JavaScript engines. We assume that both JavaScript and Wasm code are untrusted. This is a valid assumption considering that browsers run such code from potentially malicious websites.

We aim to uncover security vulnerabilities arising from the interaction between JavaScript and WebAssembly code within JavaScript engines. Despite the protections mentioned above, JavaScript engines can still crash, for example, by compiling or executing malformed JavaScript and Wasm code [8]. Crashing vulnerabilities can arise from various weaknesses in the underlying code, such as memory overflow, type confusion, and null-pointer dereferences. These vulnerabilities expose security flaws that, when exploited, can lead to severe consequences [19, 36].

2.3.2 Wasm and JavaScript Runtime Security. We review existing work and explain how our work differs from others.

Vulnerabilties in Wasm Runtimes. Wasm runtime is a potential target for attacks. Wasm runtimes might crash when compiling or executing Wasm bytecode. Recent studies have generated random Wasm bytecode to test whether Wasm runtimes can compile and execute it correctly [35, 47] or without crashing [47, 54].

Vulnerabilities in JavaScript Engines. Detecting vulnerabilities in JavaScript engines is a longstanding topic [33, 39–41, 48, 52, 53]. Similarly, these studies generate random JavaScript code to test JavaScript engines, with more recent efforts focusing on the JIT component [33, 39, 52].

This work differs from prior research by addressing a gap: earlier studies focused on generating code in a single language to uncover runtime vulnerabilities, but overlooked attack vectors arising from the interaction between JavaScript and Wasm. For example, JavaScript and Wasm can call functions from each other. But when the context switches between JavaScript and Wasm, it is likely to suffer from the classical and highly exploitable *type confusion* vulnerabilities. As our evaluation confirms, improper type handling and conversions between the two can introduce severe vulnerabilities (see §5). Additionally, Wasm and JavaScript can operate on shared resources, such as tables, which contain function pointers and are sandboxed. Whether JavaScript engines correctly handle these shared resources is another critical area that requires testing.

3 DESIGN

In this section, we present Mad-Eye, a tool that generates two-dimensional, interdependent inputs, *i.e.*, JavaScript and Wasm code, to test JavaScript engines.

3.1 Overview

The workflow of Mad-Eye is shown in Figure 1. On the Wasm side, we introduce a *wasm-generator* to generate diverse Wasm code. Unlike other Wasm fuzzers that focus only on standalone

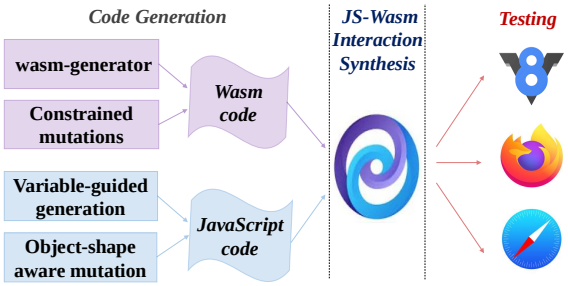

Figure 1: Overview of Mad-Eye.

Wasm code, *wasm-generator* generates Wasm modules supporting JS-Wasm interaction. Additionally, Mad-Eye applies *constrained mutations*, guided by regression tests, to harness valuable patterns that trigger vulnerabilities. On the JavaScript side, Mad-Eye generates JavaScript code that integrates with Wasm to form complex test cases. To achieve this, we design a *Variable-Guided Generation* approach that prioritizes incorporating Wasm resources in JavaScript code. Furthermore, Mad-Eye incorpotrates a fine-grained type system on top of existing mutators to enhance the accuracy of mutating Wasm objects. We call the mutation *Object-Shape Aware Mutation*. Finally, we elaborate on the *JS-Wasm Interaction Synthesis*, which bridges the cross-language gap and enables *automated* interactions between JavaScript and WebAssembly through generated WebAssembly objects and WebAssembly JavaScript APIs.

3.2 Wasm Code Generation

We discuss the generation strategy of Wasm code in this subsection.

3.2.1 Schemes. Mad-Eye employs a hybrid scheme for generating Wasm code. It *randomly* switches between the generative and mutation modes. In generative mode, Mad-Eye generates code to explore a broad range of Wasm features, aiming to maximize JavaScript engine code coverage related to Wasm (see §3.2.2). It creates a new test case by generating Wasm code. In the mutation mode, Mad-Eye randomly selects a test case from the corpus and applies *constrained mutations* (see §3.2.4) on it. Mad-Eye has an initial corpus of Wasm regression tests collected from JavaScript engine repositories. This corpus serves as the foundation for mutation and provides knowledge about vulnerability-triggering code patterns.

All test cases, whether generated or mutated, that expose new Wasm code coverage are added back to the corpus. This approach allows test cases from different sources, such as generated code and regression tests, to be combined and further mutated, resulting in new, interesting test cases.

3.2.2 Generative Approach for WebAssembly Code. There are some methods to generate random Wasm code for testing [2, 16, 35]. However, they are insufficient because, to the best of our knowledge, none of these tools support generating import and export interfaces for interacting with JavaScript. To address these limitations, we design a *wasm-generator* with several new features, including the ability to 1) test across various JavaScript engines and 2) generate Wasm modules that enable interaction with JavaScript environments.

V8's Wasm Fuzzing Driver. Mad-Eye's *wasm-generator* leverages a libFuzzer fuzzing driver [2], developed and actively maintained by V8, to generate a diverse range of Wasm modules. Among many existing Wasm generators [35, 47, 54], V8's Wasm fuzzing driver has certain strengths. First, it supports various new Wasm features, such as SIMD, WasmGC, and multiple return values within a single `function/block` [3]. Second, it can generate a wide range of Wasm instructions. For instance, while some tools have difficulty correctly invoking `btselect128` due to the need for three `I128` values on the stack [47], V8's fuzzing driver already covers it by a stack-based generating manner. Also, the frequent updates to V8 allow Mad-Eye to easily maintain complete support of Wasm features.

Enabling Interaction and Generalization. However, besides not supporting interaction with JavaScript, V8's fuzzing driver is also limited to testing only V8 due to using V8-specific internal APIs for compiling and executing Wasm modules. Therefore, we improve it to a general *wasm-generator* for Mad-Eye. Specifically, we intercept the Wasm bytecode produced by V8's fuzzing driver and wrap it using the `WebAssembly.Module()` API, which allows it to be executed by other JS engines. More details of generalizing Wasm bytecode generation are illustrated in §4.2. Next, we describe the details of supporting JS-Wasm interaction in *wasm-generator*.

JavaScript and Wasm interact through *imported* and *exported objects* (including functions). In Wasm, these objects are declared in the import and export sections, respectively. To enable interaction *on the Wasm side*, *wasm-generator* first randomly generates different types of import and export declarations in the Wasm module. For imports, *wasm-generator* randomly generates declarations of `Global/Memory/Table/Tag/Function` objects. Once the declarations exist, when generating Wasm opcodes, *wasm-generator* consumes (access or modify) these objects based on their types, achieving the interaction with resources from the JavaScript environments. Notably, imported variables do not differ from internal Wasm objects when being consumed. For example, when an `i32` type of variable is needed, an `i32` type of `Global` variable might be selected if it exists, regardless of whether it is declared in the import section or defined in the internal Global section.

A test case with a Wasm module containing the import section, but lacking the corresponding JavaScript code, cannot be compiled by JavaScript engines. JavaScript code must define and link these imports to the Wasm module (*e.g.*, `imports` in line 27 of Listing 2). This creates another essential obstacle for existing approaches for supporting JS-Wasm interaction. Even if they generate import sections, the Wasm modules remain non-functional without the necessary JavaScript code to define and link the imports. We address this challenge in §3.3.1.

For exports, however, objects declared in the export section must have existing definitions in other Wasm module sections. For example, to export a `Global` object from Wasm to JavaScript, that object must be defined either in the internal global section or declared in the import section (which means an object is imported from and re-exported back to JavaScript). Therefore, after generating the import and internal object sections, *wasm-generator* randomly selects some objects to place in the export section. The exports are accessible through the `exports` property of the `WebAssembly.Instance()` object. To use them, the JavaScript code must understand the details of the `exports` property. This challenge is addressed in §3.4.

3.2.3 Unification of Wasm and JavaScript Representation. After Mad-Eye obtains the Wasm bytecode from *wasm-generator*, it disassembles [1] bytecode into WasmBuilder API calls. For example, the API calls from lines 2 to 11 in Listing 2 represent the disassembled output. When being executed, WasmBuilder API calls reconstruct the Wasm module step-by-step into bytecode, which is then wrapped by the API `WebAssembly.Module()` to compile and execute.

Disassembling Wasm into the WasmBuilder format makes it easier to integrate Wasm regression tests and allows for more flexible module mutations. First, using WasmBuilder API format enables direct reuse of code patterns from regression tests. For example, copying the line `builder.addFunction(..).addBody(..)` from an existing regression and pasting it into a new test case allows for migrating the Wasm function into a new Wasm module. Besides, Mad-Eye programmatically modifies Wasm structures by embedding JavaScript code into WasmBuilder APIs. For example, Mad-Eye wraps WasmBuilder APIs in loops to effortlessly generate Wasm code repeatedly, a capability that V8's fuzzing driver lacks.

3.2.4 Mutation of WebAssembly Code. Mad-Eye mutates WasmBuilder APIs to integrate Wasm regression tests and utilize their vulnerability-triggering patterns. Among various language mutation methods [15], we use only *splicing*, as other mutations are likely to invalidate WasmBuilder APIs. For example, changing the `"env"` in line 4 of Listing 2 to another string would cause the Wasm module to fail to build, thus invalidating the entire test case. Besides, since *wasm-generator* already generates a wide range of Wasm instructions, we avoid mutating them but combine different ones.

Splicing involves copying a self-contained part of one program into another to combine features from different programs. Self-contained programs are those that are complete in both control flows and data flows and do not require external code to function. To achieve this, we adopt the strategy used by Fuzzilli's splicing mutator [39]. We randomly select a statement and perform backward *data-flow* analysis to identify the necessary statements for forming a valid splice. After that, we place the spliced program in a selected position in the host program.

We address the following constraints to avoid generating invalid test cases caused by incorrect splicing.

Avoiding Duplicate Load. We avoid splicing any `load()` statement. Each WasmBuilder program contains only one `load()` statement (*e.g.*, line 2 of Listing 2). Splicing the `load()` statement into another program would lead to duplicate `load()` statements that render the program invalid.

Code Placement After Load. We always place the spliced code after the `load()` statement. This is because WasmBuilder APIs are defined within `"wasm-module-builder.js"` file and can only be accessed after it has been loaded.

Avoiding Duplicate WasmModuleBuilder. A program calls `WasmModuleBuilder()` to invoke WasmBuilder APIs. We avoid including `WasmModuleBuilder()` in the splice, as it reduces efficiency by introducing multiple separate builders. However, this can be problematic if other statements in the splice depend on the return values

of `WasmModuleBuilder()`. To address this, we remap all builder uses in the splice to the builder used in the host program.

Always Adding Function Body. We always include the `addBody()` method in the splice whenever we splice the `addFunction()` method. Lines 5-6 in Listing 2 show an example. WasmBuilder APIs provide the `addFunction()` method to declare function names and parameters, and the `addBody()` method to specify the function body. Omitting the `addBody()` method while only adding `addFunction()` will result in a Wasm compilation error. Since these two APIs lack a Define-and-Use data flow relationship, Mad-Eye addresses this by splicing from `.addBody()` instead of `.addFunction()`.

Always Adding Unique Exports. Using exports requires them to be defined. Existing data-flow analysis cannot map export usages (line 30 in Listing 2) to their definitions (lines 5-11 in Listing 2) because they have no explicit data-flow relations. Thus, regular splicing might miss these API calls even if the exports are used. To address this, we conservatively add all exports' definitions during its backward traversal.

Besides missing export definitions, duplicated export definitions cause compilation exceptions in the Wasm environment. We avoid this by using the combination of the export name (*e.g.*, `add()`) and type (*e.g.*, `Function`) to identify and skip duplicates.

Skipping Incorrect Imports. Wasm places the declarations of imported functions or objects in a section (*i.e.*, the import section) that precedes the sections for internal functions or objects. Therefore, WasmBuilder APIs that declare imported objects (*e.g.*, `addImportedGlobal()`) will throw exceptions if called after APIs that declare internal objects (*e.g.*, `addGlobal()`). These errors can easily be introduced during splicing. To mitigate this, we adjust WasmBuilder API implementations to skip incorrect imports instead of throwing exceptions that halt test case execution. This ensures that imports are emitted correctly while maintaining flexibility for mutating other code.

3.3 JavaScript Code Generation

In addition to Wasm generation, Mad-Eye also generates JavaScript code that influences Wasm code executions. This includes JavaScript code that accesses imported/exported objects or JavaScript functions imported into Wasm. JavaScript code has diverse structures and complex input spaces, which can be used to trigger intricate interaction behaviors with Wasm.

Mad-Eye leverages Fuzzilli [39], a JavaScript fuzzer with diverse code generators and mutators, to construct varied and complex code structures. However, it is a new challenge to generate JavaScript code that is both *diverse* and *relevant* to Wasm features for detecting deep Wasm vulnerabilities. Directly using Fuzzilli would cause significant distraction as it tends to test various components of the JavaScript engine rather than focusing on Wasm. To address this issue, we propose the following strategies to guide Mad-Eye in generating and mutating JavaScript code.

3.3.1 Variable-Guided Generation for Pure JavaScript Code. Mad-Eye guides the code generators to produce JavaScript code interacting with Wasm modules. The implementation of code generators typically involves two main components: 1) generating variables, which may include both primitive and composite data types, and 2) generating code or more complex composite data type variables that leverage the previously generated variables. Therefore, Mad-Eye first discovers (detailed in §3.4), generates, and remembers available WebAssembly object variables in step 1. During step 2, Mad-Eye concentrates on the selection of discovered WebAssembly objects over random selection of any JavaScript variables. For example, when generating new JavaScript object definitions or function calls, Mad-Eye will prioritize the selection of Wasm objects as their elements, arguments, or callees. Using Wasm variables to guide the code generator facilitates complex JavaScript-Wasm interactions, resulting in the following effects:

Defining WebAssembly Objects for Imports. Mad-Eye introduces new code generators to create WebAssembly objects through JavaScript API (*e.g.*, `new WebAssembly.Memory()`). After *wasm-generator* generates a Wasm module that declares imported objects, Mad-Eye ensures that the objects are defined with the correct type and arguments (*e.g.*, the initial size of a `WebAssembly.Memory` object defined in JavaScript should be greater than or equal to the `Memory` object's initial size declared in Wasm). During module instantiation, the objects are then properly linked into the module. Mad-Eye also remembers them for integrating into later interaction.

Generating JavaScript Functions for Imports. Imported functions to Wasm modules can have arbitrary code structures as they are pure JavaScript code. Guided by Wasm variables, Mad-Eye generates interesting code patterns such as harnessing WebAssembly objects inside a JavaScript function that is later imported into the WebAssembly module, testing the interoperability between JavaScript and Wasm.

Interacting with WebAssembly Objects/Functions Through Imports/Exports. Mad-Eye generates JavaScript code with diverse structures and constructs to manipulate WebAssembly objects. For example, WebAssembly objects and functions might be accessed in generated JavaScript loops, enabling the JavaScript JIT engines' interaction with the WebAssembly runtime. Additionlly, when generating function calls, Mad-Eye prefers available exported WebAssembly functions from Wasm modules, using both valid and malformed arguments to test their behaviors.

3.3.2 Mutation of Pure JavaScript Code. For JavaScript code mutation, Mad-Eye utilizes a diverse set of mutators, in constrast to WasmBuilder's splicing-only approach described in §3.2.4. Additionally, we introduce an *Object-Shape Aware* approach to enhance overall mutation effectiveness targeting JS-Wasm interactions.

Adaptive Mutation Strategies. In one test case, Mad-Eye applies distinct mutation strategies based on whether the code is pure JavaScript or a WasmBuilder API. Since WasmBuilder APIs are also written in JavaScript, Mad-Eye must differentiate them from the pure JavaScript code to apply the appropriate mutations. To achieve this, Mad-Eye places pure JavaScript code at the end of the test case, separate from the WasmBuilder APIs, which are positioned at the top. Mad-Eye identifies the WasmBuilder APIs as the code between the first `new WasmModuleBuilder()` and the last `new WebAssembly.Instance()` API calls. Any code outside this range is treated as pure JavaScript.

Mad-Eye's mutation strategies adapt to the identified regions for WasmBuilder APIs and pure JavaScript. Specifically, Wasm

mutation constraints in §3.2.4 are confined to the WasmBuilder APIs region, while more diverse mutators can be applied in the pure JavaScript code region. For example, some mutators modifies JavaScript code by changing operations (*e.g.*, replacing addition with subtraction) or adjusting variable values based on their types [39, 45, 48]. On top of these mutators, Mad-Eye is further equipped with the following techniques for testing Wasm, as detailed below.

Object-Shape Aware Mutation. Mad-Eye extends existing type systems to enable more precise mutations of JavaScript code specifically for testing Wasm features. Existing works perform type analysis to select variables for mutation and substitute variables of the same broad type (*e.g.*, number, function) [39, 45, 48]. This coarse-grained approach is insufficient for Wasm, as it does not differentiate the unique structures and properties of Wasm objects. To address this, Mad-Eye tracks fine-grained type information by maintaining the *shapes* of Wasm objects [13] (details are provided in §3.4). This shape information includes specific property names such as `get`, `set`, and `length` for `WebAssembly.Table` objects. Mad-Eye identifies unknown objects with matching shapes as specific Wasm objects and applies shape-aware substitutions on them.

Interleaving Mutation with Pure JavaScript Generators. JavaScript mutation and generation are interleaved to create complex test cases. Mad-Eye allows inserting JavaScript control structures into Wasm modules. For example, a JavaScript loop generated by code generators might wrap some WasmBuilder APIs (*e.g.*,, `addType()`) spliced from another test case, resulting in repeated invocations of those APIs to programmatically modify the Wasm module (*i.e.*, appending a large number of types in the module). In our experiments, such a code pattern directly triggers a new vulnerability in JavaScriptCore.

3.3.3 Avoiding Side-Effects of Executing Pure JavaScript. While using these strategies, we must ensure that the pure JavaScript code does not introduce unintended side effects. Mad-Eye leverages hundreds of JavaScript code generators [11] to produce diverse JavaScript structures, facilitating extensive and complex interactions with Wasm objects. However, many of these generators are not always relevant. Since Mad-Eye is coverage-guided, it may be affected by coverage noise from executing irrelevant JavaScript code. To mitigate this, we focus coverage recording exclusively on Wasm implementations within JavaScript engines. Details on identifying these Wasm implementations are provided in §4.1. This directs the mutations and fuzzing efforts towards exploring Wasm features without interference from irrelevant JavaScript code [46].

3.4 Enabling JavaScript-Wasm Interactions

Wasm/JavaScript must know the names and types of imported/exported functions or objects before they can be used. This presents a challenge due to the cross-language nature, as functions or objects defined in Wasm/JavaScript are not natively presented and understood by JavaScript/Wasm. To address these issues, we employ an instrumentation-based approach named *probing*. The details of this approach are presented in Algorithm 1.

Probing Exports. We implement a *Probe()* utility that instruments a JavaScript built-in `Object.getOwnPropertyNames()` call on an object to extract its property names. Mad-Eye uses the utility on the

Algorithm 1: The Probing Algorithm

```
1  Require: A test case js_program containing Wasm objects.
2  ▷ Probe(object) : A utility that returns the instrumentation
     code to get property names of object
3  DA ← DataFlowAnalyzer (js_program)
4  instrumented_js_program ← ∅
5  interesting_sources ← ∅
6  for instr ∈ js_program do
7      instrumented_js_program.insert (instr)
8      // A GETPROP instr in JS: output = input.propertyName
9      if instr.type ≠ GETPROP then
10         continue;
11     // Probing:
12     if instr.propertyName == "exports" then
13         instrumented_js_program.insert (Probe (instr.output))
14         interesting_sources.insert (instr.output)
15     // Instrument a in cases like a = WebAssembly.Global
16     if instr.propertyName ==
        "Global" || instr.propertyName ==
        "Memory" || instr.propertyName ==
        "Table" || instr.propertyName == "Tag" then
17         interesting_sources.insert (instr.output)
18     def_operand ← DA.define_by (instr.input)
19     // Instrument c in cases like a = b.exports; c = a.mem
20     if interesting_sources.contains (def_operand) then
21         instrumented_js_program.insert (Probe (instr.output))
22 // Store probing results for JS code generation and mutation:
23 probing_results ← execute (instrumented_js_program)
```

`exports` property of objects in `WebAssembly.Instance` (*e.g.*, `wasmInstance.exports` in Listing 1). This enumerates the names of exported Wasm resources, as they can only be accessed through the `exports` property. The dynamically retrieved names are sent back to Mad-Eye, enabling it to generate JavaScript code utilizing exported resources.

In addition to probing `exports`, this process is recursively applied to previously discovered objects (e.g., `exports.mem`). One challenge is automatically identifying previously explored objects among the variables in a test case. To address this, we implement a simple data-flow analysis. Mad-Eye performs forward data-flow analysis starting from the `exports` property to track its data dependencies (lines 19-22 in algorithm 1). This analysis identifies `exports`'s derived variables that represent WebAssembly objects. For example, in the instruction sequence sliced by data-flow analysis `"v1=wasmInstance; v2=v1.exports; v3=v2.mem;"`, `v3` is recognized by Mad-Eye as a WebAssembly object. Since Mad-Eye builds on Fuzzilli, which translates JavaScript into an Intermediate Representation using Static Single Assignment [39], this data-flow analysis conducted at the IR level is necessary. Once these exported WebAssembly objects are identified, Mad-Eye performs *probing* on them again to uncover interactions.

Probing Imports. In addition to exports, Wasm interacts with JavaScript through imports. Imported resources can be defined using Wasm objects (e.g., `WebAssembly.Global` in Listing 2). To identify

potential variables representing imported WebAssembly objects, Mad-Eye conducts a similar data-flow analysis starting from the `WebAssembly` objects. It probes their derived object variables (lines 18-19 in algorithm 1), which include those imported as arguments to the `WebAssembly.Instance()` constructor. For example, in the code snippet `"v1=WebAssembly; v2=v1.Table; v3=new v2()"`, `v3` is identified as a potential imported object. Mad-Eye probes `v3` to identify the available methods that can be invoked to manipulate imports.

4 IMPLEMENTATION

Mad-Eye is open-sourced at https://anonymous.4open.science/r/Mad-Eye-1B2E. Mad-Eye is implemented on top of Fuzzilli, with 2,717 new or modified lines of Swift code; *wasm-generator* is implemented with 1,177 new or modified lines of C++ code, based on V8's WebAssembly fuzzing driver and the WasmBuilder disassembler. We discuss some important implementation details below.

4.1 Partial Instrumentation

Since JavaScript engines are large, conventional coverage-guided fuzzing can easily be diverted from testing the WebAssembly part of the engine. To address this, we only instrument the WebAssembly-related code in the JavaScript engines to guide Mad-Eye during fuzzing. Specifically, we manually inspect the engines' source code structure and identify files related to WebAssembly by checking if the folder or file names contain "wasm" or "WebAssembly". We then modify the build system configurations of each engine (e.g., ninja rules for V8 and JavaScriptCore, CMake flags for SpiderMonkey) to enable coverage feedback instrumentation exclusively for these files. The proportion of instrumented code blocks relative to the total code blocks in V8, SpiderMonkey, and JavaScriptCore are 7.33%, 9.34%, and 8.32%, respectively.

4.2 Engine-Specific Customizations

V8, SpiderMonkey, and JavaScriptCore all execute Wasm code, but differences in their support for Wasm affect our code generation strategy. First, JavaScriptCore supports only a single memory section (either imported or internal) per Wasm module, while V8 and SpiderMonkey allow multiple memory sections. As a result, Mad-Eye only generates one `Memory` when fuzzing JavaScriptCore. Second, the JavaScript built-in functions that can be imported into a Wasm module vary across engines. Specifically, V8 and SpiderMonkey support JavaScript string built-in functions for use in Wasm code, while JavaScriptCore does not support any. The detailed differences in Wasm support across JS engines are described in [22]. To ensure the accuracy of the generated test cases, Mad-Eye accounts for these engine-specific differences when creating fuzzing inputs.

5 EVALUATION

In evaluation, we answer three research questions:

- RQ1: Is Mad-Eye effective at identifying vulnerabilities in JavaScript-WebAssembly attack surfaces?
- RQ2: How does each component in Mad-Eye contribute to its performance of code coverage and vulnerability detection?
- RQ3: How does Mad-Eye compare to existing approaches?

5.1 RQ1: Performance of Mad-Eye

We evaluate Mad-Eye's capabilities in detecting vulnerabilities in JavaScript engines.

5.1.1 Settings. We first introduce our experimental settings.

Target Engines. We selected V8, SpiderMonkey, and JavaScriptCore as our fuzzing targets, as these are the JavaScript engines powering mainstream browsers: Chrome, Firefox, Safari, and Internet Explorer. We did not test standalone JavaScript engines or Wasm runtimes. To build the three engines, we used the same command-line flags as Fuzzilli [39], with additional Wasm-specific flags included. All engines were tested using their latest versions available at the time of the experiments (November 2024). We also evaluated ablated tools and other works on the same engine versions, unless otherwise specified.

Fuzzing Time. We ran Mad-Eye on each engine for 90 CPU days to detect vulnerabilities. Mad-Eye is built on Fuzzilli [11] and leverages its distributed fuzzing utilities, *i.e.*, multiple fuzzing instances share their state, including code coverage and interesting inputs. For distributed fuzzing, we utilized 30 cores per engine running concurrently for three days, totaling 90 CPU days (30 cores × 3 days). This time budget is reasonable, as prior studies have required several CPU-months or even CPU-years to uncover vulnerabilities in JavaScript engines [37, 47, 48, 52].

Experimental Environments. All experiments were conducted on a server equipped with Intel Xeon Platinum 8450H processors, 1TB of RAM, and an x86_64 architecture. V8 and SpiderMonkey provide simulators that allow simulating non-x86_64 architectures on an x86_64 machine. We used these simulators to test vulnerabilities in these two engines on other architectures (*i.e.*, arm64, mips64, riscv64, loong64). However, unless specifically mentioned otherwise, we conducted the ablation and comparative study on x86_64 across all engines.

5.1.2 Results. We list the vulnerability detection results of Mad-Eye in Table 1. After duplications, Mad-Eye detected 7, 4, and 9 previously unknown vulnerabilities in V8, SpiderMonkey, and JavaScriptCore, respectively. After manual triage, we find that Mad-Eye generates pure JavaScript code to trigger two vulnerabilities, yet most (18) of the vulnerabilities it detected are related to WebAssembly features. As shown in the `"Vul. Type"` column, 13 vulnerabilities are caused by the interactions between JavaScript and Wasm code, and 5 vulnerabilities are triggered when compiling Wasm modules.

While all these vulnerabilities result in crashes in JavaScript engines, their security implications vary. The first factor influencing severity is the relevance of the issue to real-world scenarios. For example, V8 developers prioritize vulnerabilities in core components over simulator-related issues, which are typically tied to architecture-specific code generators and are considered less severe. The second factor is the phases of error. For example, vulnerabilities in the Wasm module compilation are less critical than those triggered during the execution of Wasm modules in JavaScript (*i.e.*, the `JS-Wasm` vulnerability type) because the latter are usually easier to achieve exploitation [7, 30]. As of writing, developers have confirmed 17 vulnerabilities and fixed 11. Three unconfirmed assertion failures in JavaScriptCore are pending investigation because of

Table 1: Unique, previously unknown vulnerabilities identified by Mad-Eye.

#	JS	Issue ID	Architecture	Type	Status	Descriptions
1	V8	anon	x86_64	JS-Wasm	Fixed	Illigal write due to overwritting 'GetMemOp' operand register
2	V8	anon	x86_64	JS-Wasm	Fixed	Type confusion due to leaking relative types from canonicalizer
3	V8	anon	x86_64	JS-Wasm	Fixed	Miss checks of shared element types
4	V8	anon	arm64	JS-Wasm	Confirmed	Debug check failure when simulating stack overflow
5	V8	anon	mips64	JS-Wasm	Fixed	Wrong register conflict check
6	V8	anon	mips64	JS-Wasm	Confirmed	Unsupported JavaScript Promise Integration APIs
7	V8	anon	riscv64	JS-Wasm	Confirmed	Debug check failure in assembler-riscv.cc
8	SpiderMonkey	anon	x86_64	JS-Wasm	Fixed	Miss serialization check when importing builtin functions
9	SpiderMonkey	anon	loong64	Wasm	Fixed	Register allocation conflicts in LIR and atomic operations
10	SpiderMonkey	anon	mips64	JS	Confirmed	Assertion failure in Simulator-mips64.cpp
11	SpiderMonkey	anon	loong64	JS	Confirmed	Assertion failure in WarpOracle.cpp
12	JavaScriptCore	anon	x86_64	JS-Wasm	Fixed	Null pointer dereference in WasmOperationsInlines.h
13	JavaScriptCore	anon	x86_64	Wasm	Fixed	Miss 'exnref' type handling
14	JavaScriptCore	anon	x86_64	JS-Wasm	Confirmed	Assertion failure in WasmTypeDefinition.h
15	JavaScriptCore	anon	x86_64	JS-Wasm	Fixed	Assertion failure in WasmBBQJIT.cpp
16	JavaScriptCore	anon	x86_64	JS-Wasm	Fixed	Assertion failure in WasmBBQJIT.cpp
17	JavaScriptCore	anon	x86_64	Wasm	Reported	Assertion failure in JSWebAssemblyInstance.cpp
18	JavaScriptCore	anon	x86_64	Wasm	Reported	Assertion failure in WasmTable.cpp
19	JavaScriptCore	anon	x86_64	Wasm	Reported	Assertion failure in WasmTypeDefinitionInlines.h
20	JavaScriptCore	anon	x86_64	JS-Wasm	Fixed	Assertion failure in ObjectPropertyConditionSet.cpp

```
1  load('test/mjsunit/wasm/wasm-module-builder.js');
2  const builder = new WasmModuleBuilder();
3  const type = builder.nextTypeIndex();
4  builder.addType(makeSig([], [wasmRefType(type)]));
5  const func2 = builder.addFunction('func2', type);
6  func2.addBody([kExprRefFunc, func2.index]).exportFunc
       ();
7  const instance = builder.instantiate();
8
9  // JS-Wasm interactions
10 instance.exports.func2();
11 const v200 = instance.exports.func2();
12 v200.toString(); // V8 crashes due to type confusion
       of the exported function
```

Listing 3: A type confusion in V8.

their lower exploitation potential. Some confirmed vulnerabilities specific to certain architectures (*e.g.*, loong64) have not been fixed, as their fixes require collaboration with external developers, which is still in progress.

5.1.3 Case Studies. We discuss some new vulnerabilities detected by Mad-Eye to showcase its efficiency.

A Type Confusion in V8. Listing 3 demonstrates a minimized proof-of-concept for a type confusion vulnerability in V8. This vulnerability existed in the Chrome stable version and was marked as *high severity* by V8 developers.

The PoC defines a self-referential Wasm function, exports it, and causes V8 to crash when the exported function is used in JavaScript. Specifically, it defines a function signature with a return type referencing itself in line 5, then adds `func2()` with this signature and exports it in line 7. JavaScript code invokes the exported Wasm function `func2()` twice (lines 10-12). When `toString()` is called on the exported function, V8 crashes. The type confusion arises because

```
1  load("test/mjsunit/wasm/wasm-module-builder.js");
2
3  const builder = new WasmModuleBuilder();
4  const struct1 = builder.addStruct([makeField(kWasmI32,
       false)]);
5  const array1 = builder.addArray(wasmRefNullType(
       struct1));
6  const segment1 = builder.addPassiveElementSegment([],
       wasmRefNullType(struct1));
7  builder.addFunction("drop", kSig_v_v).addBody([
       kNumericPrefix,kExprElemDrop,segment1]).
       exportFunc();
8  const function_body = [kExprLocalGet,0,kExprLocalGet
       ,1,kGCPrefix,kExprArrayNewElem,array1,segment1,
       kExprLocalGet,2,kGCPrefix,kExprArrayGet,array1,
       kGCPrefix,kExprStructGet,struct1,0];
9  builder.addFunction("init_and_get", makeSig([kWasmI32,
       kWasmI32,kWasmI32], [kWasmI32])).addBody(
       function_body).exportFunc();
10 const instance = builder.instantiate();
11 instance.exports.drop();
12 instance.exports.init_and_get(); //JSC segfault here
```

Listing 4: A null-pointer-dereference in JavaScriptCore.

V8 mistakenly handled the WebAssembly type indices [12] when triggering JIT optimization of the WebAssembly function.

A Null-Pointer-Dereference in JavaScriptCore. Listing 4 presents a minimized proof-of-concept for a null-pointer-dereference vulnerability in JavaScriptCore.

The exploit is generated from an existing unit test [4]. In the original test, the Wasm module defines and exports two functions: `drop()` and `init_and_get()`. The test first calls `init_and_get()` and then invokes `drop()` to verify the behavior of `WebAssembly.Table` objects. During fuzzing, Mad-Eye utilizes its *probing* feature to discover the available exported function names. It generates test cases that produce multiple exported function calls, which ultimately results

Table 2: Coverage data of ablated tools, other works, and MAD-EYE in JavaScript engines on x86_64 architecture after 90 CPU days of fuzzing. The data represents the percentage of code blocks (regions), lines, functions, or branches covered relative to the total instrumented code.

		V8				SpiderMonkey				JavaScriptCore			
		Region	Line	Function	Branch	Region	Line	Function	Branch	Region	Line	Function	Branch
Ablated tools	wasm-generator*	36.75%	47.16%	39.29%	32.85%	51.86%	64.46%	51.87%	48.18%	51.3%	62.61%	58.37%	47.57%
	MAD-EYE-uncon	46.61%	58.01%	51.46%	42.74%	59.16%	73.8%	60.52%	55.56%	59.82%	71.28%	69.0%	55.06%
	MAD-EYE-con	**49.28%**	59.93%	**54.9%**	**46.61%**	65.8%	77.74%	68.02%	65.61%	63.72%	74.52%	72.34%	59.25%
Other works	RGFuzz	29.77%	41.29%	30.5%	24.85%	35.65%	48.74%	34.16%	30.97%	32.45%	39.37%	35.63%	30.42%
	WASMaker	10.4%	21.42%	13.58%	8.68%	26.67%	37.63%	24.69%	24.03%	17.67%	19.52%	14.75%	15.88%
	Fuzzilli	45.22%	56.57%	50.02%	41.93%	64.29%	75.57%	66.14%	63.75%	60.32%	67.47%	63.06%	54.77%
MAD-EYE		48.96%	**60.1%**	54.81%	46.39%	**66.3%**	**78.03%**	**68.48%**	**66.18%**	**64.72%**	**75.76%**	**74.29%**	**60.46%**

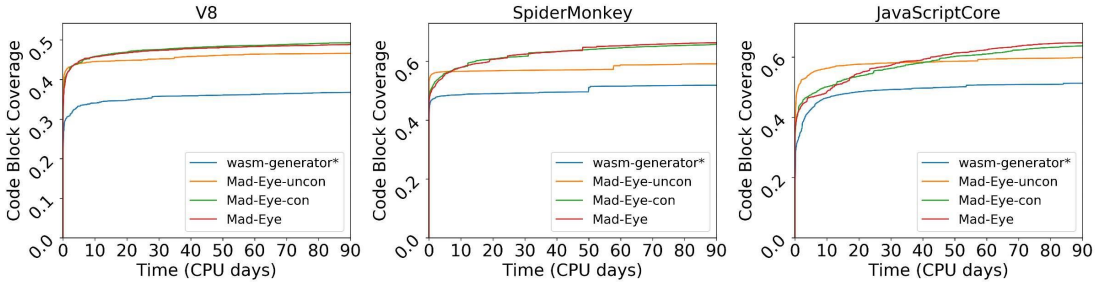

Figure 2: Code block coverage trend of ablated tools and MAD-EYE.

in calling `drop()` before `init_and_get()`. This sequence triggers a null pointer dereference because `init_and_get()` attempts to operate on null pointers that result from calling `drop()`.

5.2 RQ2: Ablation Study

In this subsection, we evaluate the effectiveness of MAD-EYE's components in detecting vulnerabilities, using the same experimental setup (*e.g.*, engine versions, hardware) described in §5.1.1. We additionally measure code coverage for the ablated tools and MAD-EYE using Clang's source-based coverage [18] during engine builds.

MAD-EYE comprises two major strategies for ablation: enabling JavaScript-Wasm interactions and constrained mutation for leveraging Wasm regressions. As MAD-EYE uses *wasm-generator* to generate Wasm code, we use *wasm-generator** as the baseline, which preserves its generalization to test different engines while ablating the design of generating imports and exports interaction interfaces. To evaluate the contributions of MAD-EYE's components, we implemented three prototypes: MAD-EYE-UNCON, which integrates Wasm regressions into *wasm-generator**'s generated code while performing unconstrained mutations; MAD-EYE-CON, which applies constrained mutations in §3.2.4 on WasmBuilder; and MAD-EYE, which enables constrained mutations and generates JavaScript code for interactions based on *wasm-generator*.

Coverage. The trends in code block coverage are presented in Figure 2, with the final coverage data detailed in Table 2. The coverage data represents the percentage of code blocks, lines, functions, or branches covered relative to the total instrumented code for each tool. While not all WebAssembly-related features may be implemented in the files we instrument in §4.1, we believe that the coverage provides a reliable indicator of the comprehensiveness of Wasm testing. *wasm-generator** has the lowest coverage among the three engines, as it employs a purely generative approach and does not leverage comprehensive regression tests. MAD-EYE-UNCON shows lower coverage than both MAD-EYE and MAD-EYE-CON due to its mutations causing Wasm compilation and runtime errors. MAD-EYE and MAD-EYE-CON show similar coverage levels throughout the experiments.

We analyzed the source-based coverage results of the instrumented files. The primary reasons for the unexecuted code are: 1) JIT optimizer for Wasm shows relatively low coverage (*e.g.*, 4,114 out of 8,212 blocks in V8's Turboshaft are not covered) since MAD-EYE is not explicitly designed to target the JIT optimization, and (2) header files contain much code but often show low coverage. Systematically exploring Wasm-JIT interactions is left for future work.

We do not show coverage data for all code in JavaScript engine repositories due to unrelated coverage noise, although we have also evaluated it. MAD-EYE achieves the highest all-code coverage because it generates pure JavaScript code, whereas the ablated tools primarily focus on generating WasmBuilder APIs.

Table 3: The number of vulnerabilities detected by ablated tools, other works, and MAD-EYE in JavaScript engines on the x86_64 architecture after 90 CPU days of fuzzing. The numbers in parentheses represent numbers of previously unknown vulnerabilities.

		V8	SpiderMonkey	JavaScriptCore
Ablated tools	*wasm-generator**	1 (1)	0	2 (1)
	MAD-EYE-uncon	1 (1)	0	3 (3)
	MAD-EYE-con	2 (2)	1 (1)	8 (6)
Other works	RGFuzz	0	0	0
	WASMaker	0	0	1 (1)
	Fuzzilli	0	0	5 (5)
MAD-EYE		**5** (3)	**1** (1)	**10** (9)

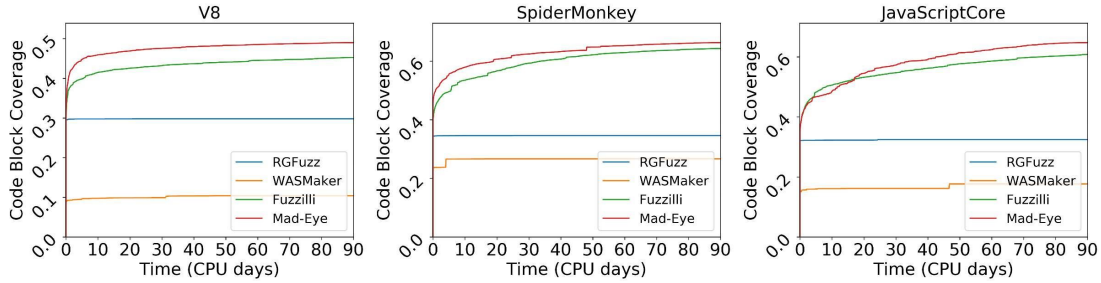

Figure 3: Code block coverage trend of other works and Mad-Eye.

Vulnerability Number. Table 3 shows the unique vulnerabilities found by the ablated tool and Mad-Eye. These include: 1) the previously unknown vulnerabilities listed in Table 1 (their numbers are noted in parentheses in Table 3); and 2) those triggered in the versions tested by Mad-Eye or the ablated tools in November 2024, but no longer reproducible in the December 2024 versions at the time of our reporting. We infer that the latter were patched by the vendors concurrently and, therefore, do not classify them as *"previously unknown"* in Table 1. Nevertheless, the total vulnerabilities demonstrate the vulnerability detection capabilities of each tool.

The results show that Mad-Eye-uncon finds the same number of vulnerabilities as *wasm-generator**, indicating that randomly integrating regressions can improve coverage but does not effectively uncover vulnerabilities. Mad-Eye-con identifies seven additional vulnerabilities compared to Mad-Eye-uncon by leveraging constrained mutation to incorporate code fragments from regressions. Through interaction synthesis, Mad-Eye further outperforms Mad-Eye-con by detecting five more vulnerabilities, all related to JavaScript-Wasm interactions.

5.3 RQ3: Comparison with Other Works

While Mad-Eye is effective at detecting new vulnerabilities, its quantitative performance compared to state-of-the-art tools remains uncertain. To address RQ3, we evaluate Mad-Eye against other state-of-the-art tools, including RGFuzz [47] and WASMaker [35], which, to the best of our knowledge, are the most recent and advanced approaches for generating Wasm code. We also compare Mad-Eye with the state-of-the-art JavaScript generator Fuzzilli [39] using the regression tests [23–25] that Mad-Eye uses. We exclude Wapplique [54] because it is not open-sourced.

5.3.1 Comparions with RGFuzz. RGFuzz generates diverse Wasm code to test Wasm runtimes. We conducted two experiments to compare with RGFuzz: the first evaluates whether RGFuzz can identify vulnerabilities in the versions tested by Mad-Eye, and the second evaluates whether Mad-Eye can uncover vulnerabilities detected by RGFuzz in the JavaScript engine versions it tested. For our comparison, we allocated 90 CPU days of fuzzing time per engine, per architecture, for both RGFuzz and Mad-Eye. We ran 30 RGFuzz instances concurrently for three days, with each instance using one CPU core. Since RGFuzz is a black-box, generation-based fuzzer with no feedback mechanism, parallel fuzzing performs similarly to serial fuzzing when operating under the same CPU threshold.

Versions Mad-Eye tested. We tested the JavaScript engines on the x86_64 architecture using the versions that Mad-Eye evaluated

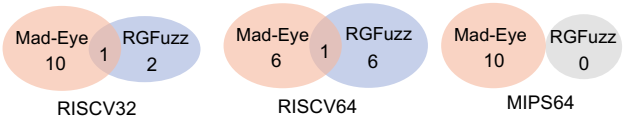

Figure 4: The numbers of bugs found by RGFuzz and Mad-Eye in the V8 version tested by RGFuzz.

(November 2024). The coverage data for Mad-Eye and RGFuzz is shown in Table 2, with their code block coverage trends presented in Figure 3. The coverage of RGFuzz is clearly lower than that of Mad-Eye. We attribute this to two main factors: First, RGFuzz employs a purely generative approach and can not leverage existing regressions. Second, RGFuzz does not incorporate coverage feedback during fuzzing. As a result, its coverage trend remains rather flat after an initial increase.

RGFuzz did not detect any vulnerabilities in our evaluation, which aligns with its previous evaluation, where it detected zero new vulnerabilities in V8, SpiderMonkey, and JavaScriptCore on x86_64 architecture after 15 CPU months of testing. In contrast, Mad-Eye detected a total of 16 vulnerabilities on x86_64 architecture after 90 CPU days of testing.

Versions RGFuzz tested. We extended our testing of JavaScript engines to three additional architectures—MIPS64, RISC-V32, and RISC-V64—where RGFuzz identified the highest number of new vulnerabilities in its evaluated version.This evaluation focused solely on V8 because we were unable to successfully build SpiderMonkey for MIPS64, RISC-V32, and RISC-V64 using the versions in the RGFuzz's repository. Additionally, RGFuzz did not detect any vulnerabilities in JavaScriptCore in its evaluation.

The detected vulnerabilities are summarized in Figure 4. Some vulnerabilities span multiple architectures and are counted individually for each architecture in Figure 4. In total, Mad-Eye identifies 18 vulnerabilities, 17 of which are not detected by RGFuzz, while RGFuzz identifies 7 vulnerabilities, 6 of which are not detected by Mad-Eye. Of the 17 vulnerabilities missed by RGFuzz, 14 involve JavaScript-Wasm interactions, and 3 are pure Wasm compilation issues. RGFuzz also identified six vulnerabilities that Mad-Eye did not find. The six vulnerabilities are pure Wasm compilation issues. To detect them, RGFuzz could be used as an alternative Wasm generator within Mad-Eye. We consider it as future work.

5.3.2 Comparions with WASMaker. WASMaker [35] is another fuzzer that generates Wasm code by disassembling and reassembling Wasm binaries. Since it does not report vulnerabilities in JavaScript engines in its evaluation, we only compare WASMaker

and Mad-Eye on the JavaScript engine versions tested by Mad-Eye. Similarly, we allocated a fuzzing time budget of 90 CPU days for both WASMaker and Mad-Eye.

Like RGFuzz, WASMaker does not leverage Wasm regressions when generating Wasm code. As shown in Figure 3, it achieves the lowest coverage among the four tools. For vulnerability detection, in Table 3, WASMaker identified one assertion failure in JavaScriptCore, which was also detected by Mad-Eye. In other words, Mad-Eye discovered 15 more vulnerabilities than WASMaker across the three JavaScript engines onå x86_64 architecture.

5.3.3 Comparions with Fuzzilli. Since Wasm regressions are written in JavaScript, we run Fuzzilli to evaluate whether existing JavaScript engine fuzzers can generate PoC exploits by mutating the Wasm regressions used by Mad-Eye. We test Fuzzilli on the versions evaluated by Mad-Eye on the x86_64 architecture.

As shown in Figure 3, Fuzzilli achieves relatively high code coverage, ranking second after Mad-Eye. Incorporating Wasm regressions [21, 23, 24] demonstrates better Wasm coverage than existing generative Wasm fuzzers during 90 CPU-day fuzzing [35, 47]. In Table 3, Fuzzilli identified five reachable assertions in JavaScriptCore, while Mad-Eye detected these five and uncovered 11 additional vulnerabilities. One reason is that Fuzzilli cannot generate new JavaScript-Wasm interactions, thus missing interaction vulnerabilities that Mad-Eye identified. These results confirm that although Fuzzilli can mutate Wasm regressions to generate Wasm and JavaScript code, it is less effective at producing new PoCs.

6 DISCUSSION

6.1 Interactions between Wasm Modules

WebAssembly modules can interact with each other by importing and exporting functions or objects. These interactions may introduce security risks. For instance, shared memory can be exploited for memory corruption or unauthorized access if not properly managed. Malicious modules could leverage such vulnerabilities to compromise data integrity or even crash the host environment. While Mad-Eye focuses on JavaScript-Wasm interactions, it usually generates one Wasm module per test case and does not properly handle inter-module linkage for interactions, therefore potentially missing vulnerabilities in those interactions. Developing automated methods to enable and test such interactions remains a direction for future work.

6.2 Testing Other JavaScript Engines

Mad-Eye can test other JavaScript engines. We currently test V8, SpiderMonkey, and JavaScriptCore because they are the major JavaScript engines embedded in mainstream browsers, which execute untrusted JavaScript and Wasm code from malicious websites. In addition, ChakraCore provides preliminary support for Wasm, and our experiments indicate it also has new security vulnerabilities related to Wasm. We do not include ChakraCore in the evaluation as it is no longer maintained or used by any mainstream browsers. Standalone JavaScript engines like JerryScript [14] and Duktape [9] are out of scope, as they currently do not support Wasm features.

7 RELATED WORK

7.1 Fuzzing JavaScript Engines

JavaScript engines are frequent targets for attacks. Finding vulnerabilities in JavaScript engines has been a significant focus of academic research. Researchers generate JavaScript code in various ways, including mutation approaches like AFL [48], deep learning [45], reinforcement learning [38], graph-based IRs [53], and language IRs [39], or generative approaches based on predefined grammars [32, 49, 50] or language IRs [37, 39]. The industry also actively tests JavaScript engines. For example, Google operates Fuzzilli for testing V8 and other engines [11]. Recently, most of these efforts have focused on the JIT component using differential techniques [33, 38, 39, 42, 44, 51, 52], which trigger JIT optimizations and check output consistency before and after JIT optimizations. Prior work has successfully reported security vulnerabilities that cause crashes or bugs that lead to unexpected outputs.

In contrast to these approaches, Mad-Eye identifies vulnerabilities in JavaScript through new attack vectors introduced by WebAssembly. It shows the importance of triggering vulnerabilities through alternative input dimensions in language interpreters.

7.2 Wasm Runtime Security

Some works focus on detecting vulnerabilities or bugs in Wasm runtimes. RGFuzz [47] implements a generative approach to detect crashes and semantic bugs in Wasm runtimes. WASMaker [35] collects and mutates real-world Wasm bytecode to conduct differential testing on Wasm runtimes. Wapplique [54] performs similar differential testing using an execution context-aware bytecode mutation. These works primarily detect semantic bugs in standalone Wasm runtimes, whereas Mad-Eye identifies security vulnerabilities in JavaScript engines.

Other works propose methods to protect Wasm and Wasm runtimes. Johnson *et al.*[43] propose a verifiably secure WebAssembly sandboxing runtime that maintains Wasm access isolation for the host OS's storage and network resources. Bosamiya *et al.*[34] propose two approaches to producing provably sandboxed Wasm code to prevent sandbox-compromising vulnerabilities in Wasm runtimes. These efforts mostly focus on standalone Wasm runtimes (*e.g.*, Wasmtime). An interesting direction would be to integrate defense mechanisms within the JavaScript engine context.

8 CONCLUSION

In this paper, we introduce Mad-Eye, the first fuzzer designed to systematically test the execution of both JavaScript and WebAssembly code within JavaScript engines. Unlike previous approaches that focus solely on generating JavaScript or Wasm code independently, Mad-Eye generates both types of code in an interdependent manner. By leveraging regression tests with constrained mutation and employing an instrumentation-based probing technique, Mad-Eye uncovers deeper, previously overlooked vulnerabilities. Our evaluation across V8, SpiderMonkey, and JavaScriptCore identified 20 previously unknown vulnerabilities, 17 of which have been confirmed, and 11 have already been fixed by browser vendors. These results emphasize the importance of testing JavaScript and Wasm executions to detect vulnerabilities in JavaScript engines.

REFERENCES

[1] 2017. V8's built-in Wasm disassembler. https://chromium.googlesource.com/v8/v8/+/refs/heads/main/tools/wasm/mjsunit-module-disassembler-impl.h.

[2] 2017. V8's internal Wasm generator. https://chromium.googlesource.com/v8/v8/+/refs/heads/lkgr/test/fuzzer/wasm-fuzzer-common.cc.

[3] 2020. Multiple Return Values in Wasm. https://github.com/WebAssembly/design/issues/937.

[4] 2022. A V8 unit test. https://sourcegraph.com/github.com/v8/v8/-/blob/test/mjsunit/wasm/array-init-from-segment.js.

[5] 2023. A Deep Dive into V8 Sandbox Escape Technique Used in In-The-Wild Exploit. https://blog.theori.io/a-deep-dive-into-v8-sandbox-escape-technique-used-in-in-the-wild-exploit-d5dcf30681d4.

[6] 2023. Getting RCE in Chrome with incorrect side effect in the JIT compiler. https://github.blog/security/vulnerability-research/getting-rce-in-chrome-with-incorrect-side-effect-in-the-jit-compiler/.

[7] 2024. Black Hat USA 2024: Achilles heel of js engines exploiting modern browsers during wasm execution. https://www.blackhat.com/us-24/briefings/schedule/achilles-heel-of-js-engines-exploiting-modern-browsers-during-wasm-execution-38540.

[8] 2024. Crash in V8. https://issues.chromium.org/issues/368114039.

[9] 2024. Duktape - embeddable Javascript engine with a focus on portability and compact footprint. https://github.com/svaarala/duktape.

[10] 2024. Emscripten. https://emscripten.org/.

[11] 2024. Fuzzilli in Google. https://github.com/v8/v8/tree/main/test/fuzzilli.

[12] 2024. Indices: WebAssembly 2.0 Specifications. https://webassembly.github.io/spec/core/syntax/modules.html.

[13] 2024. JavaScript Objects and Shapes. https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Scripting/Object_basics.

[14] 2024. Jerryscript. https://github.com/jerryscript-project/jerryscript.

[15] 2024. Mutations in fuzzing. https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html.

[16] 2024. RGFuzz repository. https://github.com/kaist-hacking/RGFuzz.

[17] 2024. The security model of WebAssembly. https://webassembly.org/docs/security/.

[18] 2024. Source based code coverage. https://clang.llvm.org/docs/SourceBasedCodeCoverage.html.

[19] 2024. A type confusion in V8 leads to RCE. https://github.blog/security/vulnerability-research/getting-rce-in-chrome-with-incorrect-side-effect-in-the-jit-compiler/.

[20] 2024. Type confusion in v8 wasm. https://issues.chromium.org/issues/332081797.

[21] 2024. A V8 regression. https://sourcegraph.com/github.com/v8/v8/-/blob/test/mjsunit/regress/wasm/regress-364667545.js.

[22] 2024. Wasm Feature Extensions. https://webassembly.org/features/.

[23] 2024. Wasm regression tests in JSC. https://github.com/WebKit/WebKit/tree/main/JSTests/wasm.

[24] 2024. Wasm regression tests in SpiderMonkey. https://github.com/mozilla/gecko-dev/tree/master/testing.

[25] 2024. Wasm regression tests in V8. https://chromium.googlesource.com/v8/v8/+/refs/heads/main/test/mjsunit/wasm/.

[26] 2024. Wasm Sandbox in JavaScript engines. https://v8.dev/blog/sandbox.

[27] 2024. Wasm Security. https://webassembly.org/docs/security/.

[28] 2024. Wasmbuilder. https://chromium.googlesource.com/v8/v8/+/refs/heads/main/test/mjsunit/wasm/wasm-module-builder.js.

[29] 2024. Wasmer. https://wasmer.io/.

[30] 2024. WebAssembly Is All You Need:Exploiting Chrome and the V8 Sandbox 10+ times with WASM. https://codeblue.jp/en/program/time-table/day2-111/.

[31] 2024. WebAssembly JavaScript APIs. https://developer.mozilla.org/en-US/docs/WebAssembly/Using_the_JavaScript_API.

[32] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for deep bugs with grammars.. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.

[33] Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. JIT-picking: Differential fuzzing of JavaScript engines. In *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS)*. Los Angeles, CA, USA.

[34] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. 2022. {Provably-Safe} multilingual software sandboxing using {WebAssembly}. In *Proceedings of the 31st USENIX Security Symposium (Security)*. Boston, MA, USA.

[35] Shangtong Cao, Ningyu He, Xinyu She, Yixuan Zhang, Mu Zhang, and Haoyu Wang. 2024. WASMaker: Differential Testing of WebAssembly Runtimes via Semantic-Aware Binary Generation. In *Proceedings of the 33th International Symposium on Software Testing and Analysis (ISSTA)*. Vienna, Austria.

[36] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. 2020. {KOOBE}: Towards facilitating exploit generation of kernel {Out-Of-Bounds} write vulnerabilities. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Virtual Event.

[37] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. 2021. One engine to fuzz'em all: Generic language processor testing with semantic validation. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[38] Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. 2024. Fuzzing JavaScript Interpreters with Coverage-Guided Reinforcement Learning for LLM-Based Mutation. In *Proceedings of the 33th International Symposium on Software Testing and Analysis (ISSTA)*. Vienna, Austria.

[39] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. 2023. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities.. In *Proceedings of the 2023 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.

[40] Xiaoyu He, Xiaofei Xie, Yuekang Li, Jianwen Sun, Feng Li, Wei Zou, Yang Liu, Lei Yu, Jianhua Zhou, Wenchang Shi, et al. 2021. Sofi: Reflection-augmented fuzzing for javascript engines. In *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS)*. Virtual Event, Korea.

[41] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Security Symposium (Security)*. Bellevue, WA, USA.

[42] Yazhuo Jin. 2024. Fuzzing for redundancy elimination vulnerabilities in just-in-time JavaScript engines. In *Fifth International Conference on Computer Communication and Network Security (CCNS 2024)*, Vol. 13228. SPIE, 391–396.

[43] Evan Johnson, Evan Laufer, Zijie Zhao, Dan Gohman, Shravan Narayan, Stefan Savage, Deian Stefan, and Fraser Brown. 2023. WaVe: a verifiably secure WebAssembly sandboxing runtime. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[44] Seungwan Kwon, Jaeseong Kwon, Wooseok Kang, Juneyoung Lee, and Kihong Heo. 2024. Translation Validation for JIT Compiler in the V8 JavaScript Engine. In *Proceedings of the 46th International Conference on Software Engineering (ICSE)*. Lisbon, Portugal.

[45] Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A neural network language {Model-Guided}{JavaScript} engine fuzzer. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Virtual Event.

[46] Changhua Luo, Wei Meng, and Penghui Li. 2023. Selectfuzz: Efficient directed fuzzing with selective path exploration. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[47] Junyoung Park, Yunho Kim, and Insu Yun. 2025. RGFuzz: Rule-Guided Fuzzer for WebAssembly Runtimes. In *Proceedings of the 46th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[48] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing javascript engines with aspect-preserving mutation. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA, USA.

[49] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA, USA.

[50] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. Montréal, Canada.

[51] Jiming Wang, Yan Kang, Chenggang Wu, Yuhao Hu, Yue Sun, Jikai Ren, Yuanming Lai, Mengyao Xie, Charles Zhang, Tao Li, et al. 2024. OptFuzz: Optimization Path Guided Fuzzing for JavaScript JIT Compilers. In *Proceedings of the 33rd USENIX Security Symposium (Security)*. Philadelphia, PA, USA.

[52] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. 2023. {FuzzJIT}:{Oracle-Enhanced} Fuzzing for {JavaScript} Engine {JIT} Compiler. In *Proceedings of the 32nd USENIX Security Symposium (Security)*. Anaheim, CA, USA.

[53] Haoran Xu, Zhiyuan Jiang, Yongjun Wang, Shuhui Fan, Shenglin Xu, Peidai Xie, Shaojing Fu, and Mathias Payer. 2024. Fuzzing JavaScript Engines with a Graph-based IR. In *Proceedings of the 31st ACM Conference on Computer and Communications Security (CCS)*. Salt Lake City, UT, USA.

[54] Wenxuan Zhao, Ruiying Zeng, and Yangfan Zhou. 2024. Wapplique: Testing WebAssembly Runtime via Execution Context-Aware Bytecode Mutation. In *Proceedings of the 33th International Symposium on Software Testing and Analysis (ISSTA)*. Vienna, Austria.