Fuzzing JavaScript Engines by Fusing JavaScript and WebAssembly

Jiayi Lin
The University of Hong Kong
Hong Kong SAR, China
linjy01@connect.hku.hk

Changhua Luo*
The University of Hong Kong
Hong Kong SAR, China
chdeluo@gmail.com

Mingxue Zhang
The State Key Laboratory of Blockchain and Data Security, Zhejiang University
Zhejiang, China
mxzhang97@zju.edu.cn

Lanteng Lin
The University of Hong Kong
Hong Kong SAR, China
lantern@hku.hk

Penghui Li
Columbia University
New York, United States
pl2689@columbia.edu

Chenxiong Qian
The University of Hong Kong
Hong Kong SAR, China
cqian@cs.hku.hk

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 using a novel cross-language code fusion technique. Evaluations of Mad-Eye on V8, SpiderMonkey, and JavaScriptCore detected 21 previously unknown vulnerabilities, with 20 confirmed and 18 fixed and merged into mainstream browsers by the developers, who acknowledged our reports with vulnerability bounties.

CCS Concepts

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

Keywords

WebAssembly; JavaScript Engines; Fuzzing

ACM Reference Format:
Jiayi Lin, Changhua Luo, Mingxue Zhang, Lanteng Lin, Penghui Li, and Chenxiong Qian. 2026. Fuzzing JavaScript Engines by Fusing JavaScript and WebAssembly. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3744916.3764551

*Corresponding author.

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/3744916.3764551

1 Introduction

JavaScript (JS) 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* from malicious websites, ensuring their security has become a paramount concern. Over the years, substantial efforts have been dedicated to testing the security of JavaScript engines. These studies have primarily focused on identifying vulnerabilities and bugs within JavaScript engines [24, 30–34, 36, 41, 44, 45, 49], 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 another input for JavaScript engines to facilitate high-performance applications, introducing new attack vectors that have yet to be fully addressed. The executions of malicious Wasm code create opportunities for exploitation, potentially leading to engine crashes [14] or, in severe cases, remote code execution (RCE) [2, 3]. Our preliminary studies shows that the share of Wasm-related regression test files in V8 grew from 3% of all regression test files in 2016 to 16% in 2017, eventually reaching 62.5% in 2024. These evolving security challenges emphasize the need to integrate the Wasm input vector into ongoing academic research to strengthen the security of JavaScript engines.

Recent works have explored generating diverse Wasm code for testing Wasm runtimes. For instance, Park *et al.* [40] proposed a reverse stack-based technique to generate random Wasm code. Zhao *et al.* [50] introduced an execution context-aware mutation approach to enhance Wasm code diversity. Similarly, Cao *et al.* [26] disassembled and reassembled real-world Wasm binaries to stress-test runtimes. In the industry, Google employs an internal Wasm generator [1] that generates various Wasm opcodes to evaluate the V8 engine's compilation and execution processes. While these approaches effectively test standalone Wasm runtimes, our analysis indicates that they are ineffective for testing Wasm execution *in the contexts of JavaScript engines.*

This work addresses the gap caused by the overlooked Wasm attack vector in JavaScript engines. In particular, while previous works have focused on generating standalone Wasm or JavaScript code, they overlook a critical aspect: the interactions between Wasm

```
1  // constructing Wasm module through WasmBuildler
2  load("test/mjsunit/wasm/wasm-module-builder.js");
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
11 // generating JS-Wasm interaction
12 const instance = builder.instantiate();
13 instance.exports.drop();
14 instance.exports.init_and_get(); //JSC segfault here
```

Listing 1: A null-pointer-dereference vulnerability (CVE-2024-54508) detected by Mad-Eye in JavaScriptCore.

and JavaScript code, which create a far more complex testing environment of JavaScript engine testing. Testing JS-Wasm interactions is crucial because, while Wasm runs in a sandbox within JavaScript engines, these interactions can potentially bypass the sandbox, leading to serious security risks. Listing 1 illustrates a minimized exploit for a new vulnerability detected by our tool in JavaScriptCore. In this case, two Wasm functions are *exported* to JavaScript code (lines 7 to 9) to be invoked. When the JavaScript code called exported Wasm functions (lines 13 to 14), the JavaScript engine crashed during handling interactions with Wasm. Because internally, the engine assessed a pointer (representing the type of a Wasm `ElementSegment`) that was set to `nullptr` during Wasm code execution, resulting in a null pointer deference. Note that JavaScript engines should execute *arbitrary* code with graceful error handling instead of crashing, which could lead to a denial-of-service or expose memory safety issues. This vulnerability is assigned a CVE and marked as *high-severity* by Apple developers. Similarly, Listing 3 demonstrates another newly discovered vulnerability by us where JS-Wasm interactions result in an exploitable type confusion in V8. It was discovered to be under active exploitation in the wild, and we were awarded $8,000 for reporting it. These examples illustrate how JS-Wasm interactions expose attack surfaces that neither language reveals in isolation. Such issues arise only when the two execution domains are combined: exporting a Wasm function into JavaScript forces the engine to bridge two distinct runtime models, translating Wasm's low-level representation into JavaScript's execution environment. Any missing checks or misaligned assumptions in this translation can lead to exploitable conditions. They indicate the intricate and error-prone nature of handling cross-language interactions, which are both challenging to implement and susceptible to security flaws. Notably, such vulnerabilities cannot be detected by existing approaches and further necessitate a new technique that bridges this gap and generates JS-Wasm interactions in systematic ways to test JavaScript engines.

Fusion is a technique that combines multiple code fragments into a single cohesive program. It shows great promise in testing graphics shader compilers [47] and SMT solvers [46]. However, to the best of our knowledge, no prior works have fused two different languages; instead, they fuse code fragments of programs in one language [46, 47]. Since major JavaScript engines, including V8, SpiderMonkey, and JavaScriptCore, execute both JavaScript and Wasm code, this work proposes the *first* technique that fuses code from JS and Wasm, enabling diverse cross-language interactions within the final program to test JavaScript engines.

Cross-language code fusion introduces several new challenges. First, while many language generators produce code in a single language, they do not account for the *interfaces* needed to call code written in a different language. As a result, their generated code cannot be directly used for fusion. To enable JS-Wasm interactions, the Wasm code's `import` and `export` sections are critical, as they define the resources required to interact with JS code. However, no Wasm generators [1, 26, 40, 50] intentionally produce the two sections. One explanation is that, once the Wasm code includes an `import` section, there must be corresponding JS code to define these imports; otherwise, the Wasm code cannot be compiled. Given this, generating `import` and `export` sections is fundamentally different from generating other Wasm sections, as the generation strategies for these sections depend on another language. Moreover, the fact that resources defined in one language are not natively understood in another makes fusion even more challenging. For instance, instead of simply generating a function and using it later (the common code generation strategy for testing other compilers and interpreters), a function defined in Wasm must be explicitly exported to JS through an export declaration. The JS code must then correctly interpret the `exports` object of a Wasm instance to use the Wasm function.

Besides, there is a lack of knowledge about which JS code is relevant to Wasm code, making it difficult to effectively generate JS-Wasm interactions. Existing JS code generators are coverage-guided and aim to produce diverse code. Directly using them for JS-Wasm fusion would degrade fuzzing performance by executing Wasm-irrelevant JS code. This not only reduces efficiency but also diverts the fuzzer's focus away from testing Wasm, the relatively newer attack surface this work aims to test.

This work proposes Mad-Eye, the new tool that employs a cross-language fusion technique to overcome the above challenges. To address the first challenge, Mad-Eye aims to target the full spectrum of interaction mechanisms through both imports and exports. For imports, Mad-Eye generates (1) declarations of random Wasm objects in the import section and (2) random Wasm code to use imported objects based on their types. For exports, Mad-Eye randomly selects objects from other sections and places them in the export section. When mutating Wasm code, Mad-Eye adheres to JS, Wasm, and JS-Wasm constraints to improve program validity. To bridge the cross-language gap, we introduce a *probing* technique on the generated Wasm code to extract information about Wasm imports and exports (*e.g.*, the exported object names and types). With these probed imports and exports, Mad-Eye generates JavaScript code in a *guided* manner to address the second challenge. Specifically, Mad-Eye introduces dedicated code generators to properly create and manipulate Wasm objects in JavaScript. To improve the

efficiency of testing vulnerabilities caused by JS-Wasm interactions, Mad-Eye employs *Variable-Guided Code Generation*, which prioritizes correctly typed Wasm objects over other variables. Mad-Eye also refines type constraints by considering object shapes rather than broader type categories (*i.e.*, `object` of Wasm objects).

We evaluate Mad-Eye on three major JavaScript engines: V8, SpiderMonkey, and JavaScriptCore. While these engines have been extensively tested by prior works and industry fuzzers, Mad-Eye found 21 previously unknown security vulnerabilities: nine in V8, four in SpiderMonkey, and eight in JavaScriptCore. 19 were related to the Wasm features and 16 were caused by direct JS-Wasm interactions, highlighting the need for testing this previously overlooked attack vector. As of this writing, developers have confirmed 20 of our reported vulnerabilities, of which 18 have been fixed. Our ablation studies show that generating JS-Wasm interactions improves not only code coverage but also vulnerability discovery. To compare Mad-Eye with state-of-the-art tools, we evaluate existing works RGFuzz [40], Fuzzilli [31], WASMaker [26], and V8's Wasm generator [1]. Mad-Eye demonstrates the best performance in Wasm-related code coverage and vulnerability discovery.

In summary, this work has the following contributions:

- We identified *previously overlooked attack vectors* imposed by the execution of WebAssembly code in JavaScript engines. To the best of our knowledge, we designed the first cross-language code fusion technique for generating JS-Wasm programs to detect vulnerabilities in JavaScript engines.
- We implemented all techniques in a tool called Mad-Eye, which is open-source and available at https://github.com/HKU-System-Security-Lab/Mad-Eye.
- Mad-Eye detected 21 vulnerabilities across three major JavaScript engines; 20 have been confirmed, and 18 have been fixed by the browser developers; 3 CVE IDs have been assigned.

2 BackGround

2.1 Wasm in JavaScript Engines

In this subsection, we explain how Wasm operates within JavaScript engines, how Wasm is represented through WasmBuilder APIs, and how Wasm interacts with JavaScript code.

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

- **Source Code to Wasm Bytecode:** There are two approaches to generating Wasm bytecode. The first uses compilers for high-level programming languages. For instance, `Emscripten` [6] translates C/C++ code into Wasm bytecode. The second utilizes V8's `WasmBuilder`, which produces Wasm bytecode through programmatic construction (detailed in §2.1.2). We use the second approach, as we use Wasm regression tests written in `WasmBuilder` as the initial fuzzing corpus.
- **Wasm Bytecode to Native Code:** A `.wasm` module is executed by a Wasm runtime embedded within the JS engine. The runtime validates the module, compiles it into native machine code, and then executes it.

2.1.2 WasmBuilder. V8 introduces the *WasmBuilder* library [19] to create Wasm modules in JavaScript. The WasmBuilder consists of various APIs to programmatically construct Wasm bytecode. Since WasmBuilder allows JS engines to construct and execute Wasm modules, it is also widely used in JS engines beyond V8. For instance, Wasm regression tests [16–18] in V8, SpiderMonkey, and JavaScriptCore are manually written by the JavaScript engine developers *in the format of WasmBuilder API calls*.

In Listing 2, lines 2-11 is an example of constructing a Wasm module via WasmBuilder APIs. The equivalent WebAssembly-Text (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. The API's implementation is defined in line 1, where it emits Wasm bytecode based on the provided API arguments. The bytecode is then passed to `Wasm.Module()` in line 24 for compilation. For example, `addImportedGlobal()` in line 4 emits bytecode that appends a declaration of a `Global` object `env::var` to the import section.

2.1.3 JavaScript-Wasm Interactions. We define JavaScript-Wasm interactions in JavaScript engines as the mechanism through which JavaScript code calls Wasm code and vice versa. Note that WasmBuilder APIs are *not* considered as part of these interactions. WasmBuilder APIs emit a sequence of binary data (which is interpreted as meaningful bytecode by Wasm runtime), but they do not provide a mechanism for invocation between JavaScript and Wasm code.

JavaScript and Wasm interact as follows: JavaScript can create objects to be imported into and used by Wasm modules (*imports*); and Wasm modules can 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 the Wasm JavaScript APIs [22] (*e.g.*, `new Wasm.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 `Wasm.Module()` constructor for compilation. The compiled module is subsequently instantiated in line 28 using the `Wasm.Instance()` constructor, enabling its execution and interaction with JavaScript. Wasm 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 by the JavaScript code.

2.2 Threat Model and Existing Works

In this subsection, we present our threat model and the motivations of this work.

```
// WasmBuilder APIs
load('test/mjsunit/wasm/wasm-module-builder.js'); // a
      library provided by V8.
const builder = new WasmModuleBuilder();
builder.addImportedGlobal('env', 'var', kWasmI32);
builder.addFunction('add', kSig_i_i)
  .addBody([
    kExprLocalGet, 0,
    kExprGlobalGet, 0,
    kExprI32Add
  ])
  .exportFunc();
/* its equivalent WAT format
(module
  (type (func (param i32) (result i32)))
  (import "env" "var" (global i32)) ;; import section
  (func $add (type 0) (param i32) (result i32)
    local.get 0       ;; Load the first parameter
    global.get 0      ;; Use imports
    i32.add           ;; Add them
  )
  (export "add" (func $add)) ;; export section
)
*/
let module = WebAssembly.Module(builder.toBuffer());
// JavaScript interactions
let glob = new WebAssembly.Global({value: 'i32',
     mutable: true});
let imports = {env: {var: glob}};
const wasmInstance = new WebAssembly.Instance(module,
     imports);
glob.value = 10;
wasmInstance.exports.add(3);  // Use exports
```

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

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

Our goal is to uncover vulnerabilities arising from Wasm code executions within JS engines. We test Wasm features in JS engines because they are relatively new, and JS-Wasm interactions remain unexplored in prior research. Crashes in JS engines can expose underlying vulnerabilities that may be exploited in subsequent attacks [13, 27]. Many fuzzers of JS engines [24, 30–34, 36, 41, 44, 45, 49], include V8's internal fuzzer [31], use crashes as indicators of vulnerabilities. Following this line of research, our objective is to generate code that intentionally causes JS engines to crash upon execution. In doing so, we challenge the design philosophy that *JavaScript engines should not crash when executing arbitrary code.*

2.2.2 New Attack Vectors in JavaScript Engines. Detecting vulnerabilities in JavaScript engines has been a major focus of research. Prior works proposed various strategies to generate random JavaScript code [31, 41, 45]. Some works also generate Wasm code [1, 26, 40, 50], which can be fed into JavaScript engines as inputs.

This work differs from all prior research by addressing a gap: earlier studies focused on generating JS or Wasm code to uncover vulnerabilities but *overlooked attack vectors arising from the interactions between JS and Wasm.* For example, JS engines can execute code where JS code calls Wasm functions. When the calling context switches, it is likely to suffer from the classical and highly exploitable *type confusion* vulnerabilities. As our evaluation confirms, improper type conversions between JS and Wasm can introduce severe security risks (see §5). Additionally, Wasm and JS code can operate on shared resources, such as *Table* that contains raw pointers. Whether JavaScript engines correctly handle these sensitive resources is also an area that is worth testing.

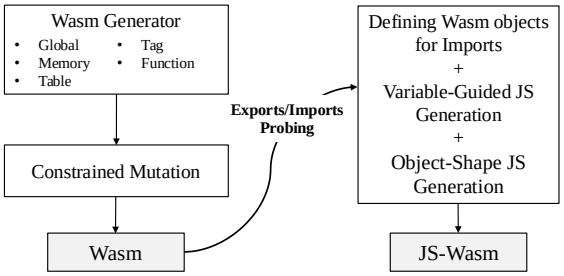

Figure 1: The workflow of MAD-EYE.

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. To generate diverse, valid, and interactive JS–Wasm programs, several challenges must be addressed. First, there exists a fundamental language gap between JavaScript and Wasm. The two languages adopt different type systems: JavaScript is dynamically typed, while Wasm enforces strict static typing. As a result, their interaction requires carefully designed interfaces through special types of objects. Existing Wasm code generators do not account for these interfaces and fail to produce programs that exercise JS–Wasm interactions. Second, while modern JavaScript fuzzers are effective at exploring large code spaces, they are typically coverage-guided in a generic manner and lack dedicated mechanisms to target JS-Wasm interaction. As a result, they tend to produce large quantities of Wasm-irrelevant code that do not contribute meaningfully to testing the attack surface we focus on.

To address them, MAD-EYE emphasizes principles of generating and mutating the full spectrum of interaction mechanisms between JavaScript and Wasm, including imports, exports, and object manipulations. To achieve this, we first design a Wasm generator that comprehensively supports interactive objects, as described in §3.2.1. On the JavaScript side, we then generate code that is aware of these objects and capable of invoking them correctly, enabled by *probing*-based discovery and type-aware construction strategies (§3.3). To further ensure both diversity and semantic validity, MAD-EYE employs principled mutation strategies on both languages: constrained mutation for Wasm that respects semantic rules (§3.2.2), and guided generation of JavaScript code that prioritizes Wasm-relevant interactions (§3.3.2).

3.1 Overview

The overall workflow of MAD-EYE is illustrated in Figure 1. At a high level, it first generates Wasm code and then constructs JavaScript code to interact with it. We adopt this generation order because JavaScript offers a vast array of features. Without a predefined

Algorithm 1: Mad-Eye Workflow

```
1  Input: Corpus C, Fuzzer F, Mutators M
2  while F.NotStopped do
3      if probability(0.5) then
4          P ← WasmGenerator()
5      else
6          P ← C.Select()
7      O ← Probe(P) ;          // extract Wasm metadata
8      for i ← 1 to Rounds do
9          m ← RandomPick(M)
10         P ← m.Mutate(P, O)
11         F ← Execute(P)
12         C.CheckSave(P, F)
```

Wasm code, it is challenging to determine which JavaScript constructs will effectively trigger and test Wasm-related functionalities. Therefore, Mad-Eye begins with a *wasm-generator* to generate Wasm modules enriched with interactive objects to be imported and exported. *wasm-generator* supports all available types for imports and exports (*Global*, *Memory*, *Table*, *Tag*, and *Function*), ensuring the presence of interaction interfaces for JavaScript. Then, Wasm modules are mutated under semantic constraints to preserve validity while increasing diversity. Next, Mad-Eye probes the Wasm module to extract metadata about imports and exports, which bridges the language gap and serves as guidance for subsequent JavaScript generation. Guided by this metadata, JavaScript code is generated that meaningfully defines and manipulates Wasm objects. Specifically, we design *Variable-Guided Generation* to prioritize the use of Wasm objects and *Object-Shape Aware Code Generation*, which leverages a fine-grained type system to improve generation accuracy.

To implement this workflow, we integrate it into a grey-box fuzzing loop (formalized in Algorithm 1) based on *Fuzzilli* [7], which is widely used in academia and industry. It starts from either selecting a previously saved program in the corpus or calling *wasm-generator* to generate a new one (lines 2 to 7). Next, Mad-Eye applies rounds of mutation through a set of mutators, including the ones we designed and others from *Fuzzilli*. In each round, Mad-Eye randomly selects a mutator and applies it to the program while being aware of the probed Wasm metadata (lines 8 to 11). The resulting program is then executed in the target engine with Wasm-related coverage fed back into the fuzzer; programs that trigger new coverage are saved in the corpus for further mutation (lines 12 to 13). We also leverage vulnerability oracles from existing works, *i.e.*, Mad-Eye considers inputs that trigger crashes in JS engines as evidence of vulnerabilities.

3.2 Wasm Code Generation

We describe the generation strategy of JS-interactive Wasm code.

3.2.1 Generating JS-Interactive Wasm Code. There are some methods to generate random Wasm code for testing [1, 26, 40, 50]. However, they are insufficient because, to the best of our knowledge, none of the existing tools can generate Wasm code that is intentionally interactive with JS. To address these limitations, we design a *wasm-generator* that can generate Wasm code equipped with diverse objects (§2.1.3) to interact with JS code.

JavaScript and Wasm interact through imported and exported objects, which are declared in the import and export sections of Wasm, respectively. For imports, *wasm-generator* randomly generates declarations of `Global`, `Memory`, `Table`, `Tag`, and `Function` objects in the import section. Once generated, *wasm-generator* produces random Wasm code to access or modify these objects based on their types. These objects, declared in Wasm's import section and defined in JS code, allow Wasm code to interact with resources from the JS code. Notably, imported objects are treated the same as internal Wasm objects when being consumed in Wasm code. For instance, if an `i32` type variable is needed, an `i32` Global variable may be selected to access, regardless of whether it's declared in the import section or defined in the internal global section.

For exports, however, objects declared in the export section must also be declared or defined in the other corresponding sections of the Wasm module. For instance, to export a Global object from Wasm to JavaScript, it must be defined in the internal global section in the Wasm module. Alternatively, it can also be declared in the import section and accessed like any other export, indicating the object is imported from and re-exported to JavaScript. After generating the import and internal sections, *wasm-generator* randomly selects some objects to place in the export section. To use these exports, the JavaScript code must understand their details, as described in §3.3.1.

3.2.2 Constrained Mutation for Wasm. Mad-Eye mutates WasmBuilder APIs to combine code from different sources, including the generated code and a collected corpus (*e.g.*, Wasm regression tests), to include vulnerability-triggering patterns. Among various language mutation methods [10], we utilize only *splicing*, which incorporates patterns between inputs while being less likely to invalidate the Wasm module compared to other random mutators.

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 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 another program. We address the following constraints in accordance with JavaScript and Wasm execution rules:

Complying with JS Constraints. As WasmBuilder APIs are in JS, they must adhere to JS execution rules. First, we avoid splicing any duplicate `load()` statement as this violate JS execution rules. Besides, we 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.

Complying with Wasm Constraints. A JS program calls `WasmModuleBuilder()` to instantiate builders before invoking WasmBuilder APIs. As each initial program has instantiated a `WasmModuleBuilder()`, we avoid including `WasmModuleBuilder()` in the splice again. 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 splice-host program.

Besides, we always include the `addBody()` method in the splice whenever splicing 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 adding `addFunction()` will result in a Wasm compilation error.

Complying with Imports/Exports Constraints. Wasm modules place the declarations of imported objects in the import section, which precedes the sections for internal 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 modify the WasmBuilder library to skip incorrect imports instead of throwing exceptions that halt code execution. This is a trade-off between diversity and validity, since it could compose more diverse inputs but might invalidate the Wasm module during its compilation.

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) because they have no explicit data-flow. To address this, we add all exports' definitions to the splice during backward traversal. Besides, duplicated export definitions would cause exceptions in the WasmBuilder. We avoid this error by using the combination of the export name (*e.g.*, `add()`) and type (*e.g.*, `Function`) to identify and skip duplicates.

3.3 JavaScript Code Generation for Interaction

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

Since there are many JS code generators, we build our JS code generator upon an existing generator, Fuzzilli [7]. Mad-Eye leverages it because Fuzzilli is actively maintained by Google. However, it is a new challenge to generate JS code that is both *diverse* and *relevant* to Wasm features for detecting more Wasm vulnerabilities. Simply using Fuzzilli would cause significant distraction as it is coverage-guided rather than focusing on the new attack vector, *i.e.*, Wasm. To address this issue, we propose the following strategies to guide Mad-Eye in generating and mutating JavaScript code.

3.3.1 Discovering Wasm Objects and Types. JavaScript must know the names and types of imported/exported Wasm objects before they can be used. This presents a challenge due to the cross-language nature, as 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*. We implement a *Probe()* utility that instruments a JavaScript built-in `Object.getOwnPropertyNames()` call on an object to extract its property names and types.

Probing Exports. Mad-Eye uses the utility on the `exports` property of objects in `WebAssembly.Instance` (*e.g.*, `wasmInstance.exports` in Listing 2). This enumerates the names and types of all exported Wasm resources, as they are only accessed through the `exports` property. In addition to probing `exports`, this process is recursively applied to previously probed objects (*e.g.*, `wasmInstance.exports.mem`). The dynamically retrieved names and types are sent back to Mad-Eye, enabling it to generate JS code utilizing exported resources.

Probing Imports. Besides exports, imports are defined in JS code using the Wasm JavaScript APIs (*e.g.*, `WebAssembly.Global` in Listing 2) and are used by the Wasm module to access JS resources. To identify potential variables representing imported Wasm objects, Mad-Eye conducts a data-flow analysis starting from `WebAssembly`. It probes the `WebAssembly` object's properties, 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()"`, Mad-Eye identifies `v3` as a potential object imported to Wasm. It then probes `v3` to identify the available methods (*e.g.*, `set()`) that can be invoked to manipulate imported resources.

3.3.2 Guided JS Code Generation. With these probed exported/imported resources, Mad-Eye generates JavaScript code in a guided manner to test Wasm attack surfaces in JS engines.

Defining Wasm objects for Imports. Mad-Eye introduces new code generators to create Wasm objects (*e.g.*, `new WebAssembly.Memory()`) through WebAssembly JavaScript API [22]. After *wasm-generator* generates a Wasm module that declares imported objects, Mad-Eye ensures that the objects are defined properly in JS code (*e.g.*, the initial size of a `WebAssembly.Memory` object defined in JS code should be greater than or equal to the `Memory` object's initial size declared in Wasm). During Wasm module instantiation, the objects are then properly linked into the module.

Variable-Guided Code Generation. Mad-Eye guides the code generators to produce JavaScript code interacting with Wasm modules. The implementation of its code generators involves two main components: 1) generating variables, which may include both primitive and composite data types, and 2) generating code or variables of composite types that leverage the previously generated variables. Mad-Eye first discovers (the discovery method is detailed in §3.3.1), generates, and records available Wasm object variables in step 1. During step 2, Mad-Eye prioritizes selecting Wasm objects over other variables to increase the likelihood of JS-Wasm interactions. For example, when generating new JavaScript object definitions or function calls, Mad-Eye prioritizes the selection of Wasm objects as their elements, arguments, or callees.

Object-Shape Aware Code Generation. Mad-Eye extends existing type systems to enable more precise construction of JavaScript code interacting with Wasm. Existing works perform type analysis to select variables for mutation and substitute variables of the same broad type (*e.g.*, number, function) [31, 37, 41]. This coarse-grained approach is insufficient for targeting Wasm, as it does not differentiate the unique structures and properties of objects. Addressing this limitation is important for testing Wasm attack surfaces as JS and Wasm interact through Wasm objects.

To address this, Mad-Eye tracks fine-grained type information by maintaining the *shapes* of Wasm objects [8]. (the *shape* of the Wasm object is probed in §3.3.1) For instance, `WebAssembly.Table` objects have properties such as `get`, `set`, etc. `WebAssembly.Memory` objects have properties like `grow`, etc. Randomly substituting these objects without considering their shapes will lead to errors such as

incorrect method calls, etc. Mad-Eye identifies objects with matching shapes as having the fine-grained same type. In this way, the JavaScript code generators and mutators can operate in a more accurate manner, such as substituting Wasm objects in the same shape to avoid invalid Wasm module compilation.

The above strategies facilitates complex and effective JS-Wasm interactions, resulting in the following effects:

- *Interacting with Wasm Objects:* Mad-Eye generates JavaScript code with diverse structures and constructs to manipulate Wasm objects. For example, it can place Wasm objects in generated JavaScript loops, triggering the JavaScript JIT optimizers to enable their interactions with the Wasm execution.
- *Generating JavaScript Functions for Interaction:* Functions imported to Wasm modules can have arbitrary code structures. Guided by Wasm variables, Mad-Eye generates interesting code patterns such as harnessing Wasm objects inside a JavaScript function that is later imported into the Wasm module, testing the interoperability between JavaScript and Wasm.
- *Interleaving Mutation within JavaScript Generators:* JavaScript code mutation is interleaved into generators, enabling complex constructions of both WasmBuilder and JavaScript regions. For example, a JavaScript loop generated in a WasmBuilder region could wrap a WasmBuilder API call, resulting in repeated API invocations to modify the Wasm module.

Avoiding Side-Effects of JavaScript Generation. Mad-Eye leverages existing JS code generators [7] to facilitate extensive and complex interactions with Wasm objects. However, the generators may not always generate Wasm-related JS code. Since Mad-Eye is coverage-guided, it may be affected by coverage noise from executing irrelevant JS code. To mitigate this, we focus exclusively on Wasm coverage within JS engines. This directs the mutations and fuzzing efforts towards exploring Wasm features without interference from irrelevant JS code [39]. Details on identifying these Wasm implementations are provided in §4.

4 Implementation

Mad-Eye is open-sourced at https://github.com/HKU-System-Security-Lab/Mad-Eye. It 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 Wasm generator and the WasmBuilder disassembler [1]. We discuss some important implementation details below.

Partial Instrumentation. Since JavaScript engines are large, conventional coverage-guided fuzzing can easily be diverted from testing the Wasm part of the engine. To address this, we only instrument the Wasm-related code in the JavaScript engines to guide Mad-Eye. Specifically, we manually inspect the engines' source code structure and identify files related to Wasm by checking if the folder or file names contain "wasm" or "WebAssembly". We then modify the build system configurations of each engine to enable coverage feedback instrumentation exclusively for these files. While not all Wasm-related features may be implemented in the files we instrument, we believe that the coverage provides a reliable indicator of the comprehensiveness of Wasm testing. 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.

Engine-Specific Customizations. We adjust the code generation strategies to adapt to the engine-specific Wasm support. First, JavaScriptCore supports only a single memory section (either imported or internal) per Wasm module, while V8 and SpiderMonkey allow multiple memory sections. Therefore, 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, whereas JavaScriptCore does not. More differences in Wasm support across JS engines are described in [15]. To improve the correctness, Mad-Eye accounts for these engine-specific differences when creating fuzzing inputs.

5 Evaluation

In the evaluation, we answer three research questions:

- RQ1: Is Mad-Eye effective at identifying vulnerabilities caused by Wasm executions in JS engines?
- RQ2: How does Mad-Eye compare to existing approaches?
- RQ3: How does each component in Mad-Eye contribute to its performance of code coverage and vulnerability detection?

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, including Chrome, Firefox, Safari, and Internet Explorer. We employed the partial instrumentation strategy (detailed in §4) to build the JavaScript engines, ran different fuzzers on them, and computed the coverage ratio by measuring the covered code relative to the total instrumented code in all experiments, including ablation and comparative studies. We built the three engines in their debug and fuzzing modes, using the developing versions available at the time of experiments[1] and the same command-line flags as Fuzzilli [31]. We also evaluated ablated tools and other works on these versions with the same compilation flags, unless otherwise specified.

Environments. We ran Mad-Eye on each engine for 90 CPU days to detect vulnerabilities, which is reasonable as prior studies also required several CPU-months or even CPU-years to uncover vulnerabilities in JavaScript engines [28, 40, 41, 45]. All experiments were conducted on a server with Intel Xeon Platinum 8450H processors and 1TB RAM. V8 and SpiderMonkey provide simulators that allow simulating non-x86_64 architectures on our x86_64 machine. We used them to test 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. In all experiments, we use the Wasm regression tests

[1]The exact commit versions of the target engines are as follows:
V8: 9422b80f49d7bb206a5d0795dc55472c6b17b161
SpiderMonkey: 4e69784010d271c0fce0927442e4f8e66ffe645b
JavaScriptCore: 1a9adbce1d3fbd78795e86aad2c57ce384e31168

```
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 func = builder.addFunction('func', type);
6  func.addBody([kExprRefFunc,func.index]).exportFunc();
7  const instance = builder.instantiate();
8  // JS-Wasm interactions
9  instance.exports.func();
10 const v200 = instance.exports.func();
11 v200.toString(); // V8 crashes due to type confusion
```

Listing 3: A type confusion vulnerability in V8.

provided in JavaScript engine repositories [16–18] as the initial fuzzing corpus, if required by the fuzzers.

5.1.2 Results. We present the vulnerability detection results of Mad-Eye in Table 1. After deduplication, Mad-Eye identified 9, 4, and 8 previously unknown vulnerabilities in V8, SpiderMonkey, and JavaScriptCore, respectively. As shown in the "`Vul. Type`" column, 15 vulnerabilities were caused by the interactions between JS and Wasm, while 4 were triggered during the compilation of Wasm. 7 vulnerabilities were linked to `Debug check failures`, which we use to reveal the causes of these issues. However, all vulnerabilities are reproducible in the stable releases of the engines. Some vulnerabilities were caused by `assertion failures`. Although assertions are typically placed in code to catch illegal cases during development, they can be considered security-relevant, as they can indicate memory errors before the assertion or lead to denial-of-service attacks. For instance, Apple assigned CVE-2025-24162 to one of them (#16).

While all these vulnerabilities result in crashes in JavaScript engines, their security implications vary. Among them, #1-#2, #7-#9, #10, #14-#18 are marked as high-severity, while the remaining are considered lower priority. The first factor influencing severity is the phases of error. 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 [4, 21]. The second factor is the relevance to real-world scenarios. For example, V8 and SpiderMonkey developers prioritize vulnerabilities in core components over architecture-specific issues. As of writing, developers have confirmed 20 vulnerabilities and fixed 18.

5.1.3 A Case Study. We discuss a new vulnerability detected by Mad-Eye to showcase its efficiency. Listing 3 demonstrates a minimized PoC for a type confusion vulnerability in V8. This vulnerability was marked as *high severity* (CVE-2024-12053) and we were awarded 8,000 USD for reporting it. The Wasm module defines a recursive type and a self-referential function `func`, which is then exported to JavaScript. When JavaScript invokes the exported function, the engine must wrap the Wasm function in a JavaScript callable object. At this boundary, V8 applies its tier-up mechanism: the first call (line 10) executes an interpreter-level wrapper, while a subsequent call (line 11) triggers tier-up to a compiled wrapper for performance. The flaw resided in V8's type canonicalizer, which incorrectly calculates type indexes in recursion groups. During tier-up, it embedded an inconsistent type index into a JavaScript function object. As a result, the method call in line 12 assessed an incorrectly typed object and crashed. The root cause lies in the translation between diverse Wasm and JavaScript types and objects during cross-language tier-up. It illustrates why JS–Wasm interaction is a risky attack surface, since every boundary crossing requires careful translation that introduces opportunities for subtle type and memory safety errors.

In Mad-Eye, the guided JS generation is the crucial step to reveal this vulnerability, especially compared to existing works. As described before, function calls in lines 11 and 12 are necessary to trigger vulnerable code in tier-up and expose the crash, respectively. To construct this PoC, Mad-Eye first generated the Wasm module either via *wasm-generator* or *splicing* from existing corpus (§3.2.2), and then probed the module to discover exported function `func`. Through probed information, Mad-Eye extensively explored JS-Wasm interaction, such as invoking discovered functions repeatedly. Furthermore, Mad-Eye would manipulate related objects through JavaScript mutators, such as objects exported by Wasm or returned by Wasm functions (line 12), to finally reveal this vulnerability.

5.2 RQ2: 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 RQ2, we evaluate Mad-Eye against other state-of-the-art tools, including RGFuzz [40] and WASMaker [26], 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 [31] using the Wasm regression tests [16–18] that Mad-Eye uses. We exclude Wapplique [50] because it is not open-sourced.

5.2.1 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.

Testing Latest-Version Engines. We tested the JavaScript engines on the x86_64 architecture using the versions that Mad-Eye evaluated. The coverage data for Mad-Eye and RGFuzz is shown in Table 3, with their code block coverage trends presented in Figure 2. 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 without incorporating initial fuzzing corpus. Second, RGFuzz does not use coverage feedback. 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 failed to detect vulnerabilities in V8, SpiderMonkey, and JavaScriptCore on x86_64 architecture after 15 CPU months of testing. In contrast, Mad-Eye detected 17 new vulnerabilities on x86_64.

Testing Old-Version Engines. 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 [11]. This evaluation focused solely on V8 because we were unable to successfully

Table 1: Unique, previously unknown vulnerabilities identified by \sys.

#	JS	Issue ID	Architecture	Type	Status	Descriptions
1	V8	378779897	x86_64	JS-Wasm	Fixed	Illigal write due to overwritting 'GetMemOp' operand register
2	V8	379009132	x86_64	JS-Wasm	Fixed	Type confusion due to leaking relative types from canonicalizer
3	V8	388685477	x86_64	JS-Wasm	Fixed	Mass instruction selection inputs lead to de-optimization crash
4	V8	379052295	x86_64	JS-Wasm	Fixed	Miss checks of shared element types
5	V8	397043084	x86_64	JS-Wasm	Fixed	Miss checks of instruction selector in Turboshaft compilation
6	V8	381917890	arm64	JS-Wasm	Fixed	Debug check failure when simulating JS stack overflow
7	V8	380604249	mips64	JS-Wasm	Fixed	Wrong register conflict check
8	V8	381325002	mips64	JS-Wasm	Confirmed	Unsupported JavaScript Promise Integration APIs
9	V8	380618369	riscv64	JS-Wasm	Confirmed	Debug check failure in assembler-riscv.cc
10	SpiderMonkey	1931471	x86_64	JS-Wasm	Fixed	Miss serialization check when importing builtin functions
11	SpiderMonkey	1938742	loong64	Wasm	Fixed	Register allocation conflicts in LIR and atomic operations
12	SpiderMonkey	1938744	mips64	JS	Fixed	Assertion failure in Simulator-mips64.cpp
13	SpiderMonkey	1933148	loong64	JS	Fixed	Assertion failure in WarpOracle.cpp
14	JavaScriptCore	283398	x86_64	JS-Wasm	Fixed	Null pointer dereference in WasmOperationsInlines.h
15	JavaScriptCore	284159	x86_64	JS-Wasm	Fixed	Tail calls should consume expression stack after call in BBQ
16	JavaScriptCore	283261	x86_64	JS-Wasm	Fixed	Miss validation of functions/exceptions signature when parsing types
17	JavaScriptCore	285065	x86_64	JS-Wasm	Fixed	WASM GC Objects should have null prototype
18	JavaScriptCore	284627	x86_64	Wasm	Fixed	Miss 'exnref' type handling
19	JavaScriptCore	284161	x86_64	Wasm	Fixed	FuncRefTable should accept JSNull when set from initElementSegment
20	JavaScriptCore	284873	x86_64	Wasm	Fixed	Assertion failure in WasmTypeDefinitionInlines.h
21	JavaScriptCore	283280	x86_64	JS-Wasm	Reported	Assertion failure in WasmBBQJIT.cpp

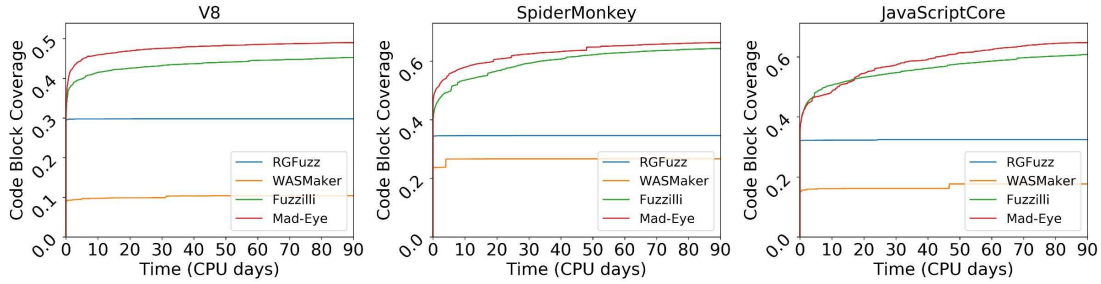

Figure 2: Code block coverage trend of other works and Mad-Eye.

Table 2: 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 (2)
	Mad-Eye-uncon	1 (1)	0	2 (2)
	Mad-Eye-Jcon	1 (1)	0	3 (3)
	Mad-Eye-Wcon	2 (2)	1 (1)	8 (6)
Other works	RGFuzz	0	0	0
	WASMaker	0	0	1 (1)
	Fuzzilli	0	0	5 (5)
Mad-Eye		**6** (4)	**1** (1)	**10** (9)

Figure 3: The numbers of bugs found by RGFuzz and Mad-Eye in the V8 version tested by RGFuzz.

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 3. Some vulnerabilities span multiple architectures and are counted individually for each architecture in Figure 3. 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 identified 6 vulnerabilities that Mad-Eye did not find. They are all Wasm compilation issues. Note that RGFuzz only detects vulnerabilities in architecture-specific code generators, demonstrating weaker real-world impact compared to Mad-Eye, which identifies vulnerabilities in core Wasm components.

5.2.2 Comparions with WASMaker. WASMaker [26] 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.

As shown in Figure 2, WASMaker achieves the lowest coverage among the four tools. For vulnerability detection, in Table 2, WASMaker identified one assertion failure in JavaScriptCore, which was

also detected by Mad-Eye. In other words, Mad-Eye discovered 16 more vulnerabilities than WASMaker across the three JavaScript engines on x86_64 architecture.

5.2.3 Comparions with Fuzzilli. Since WasmBuilder APIs are in JavaScript, the initial corpus used by Mad-Eye consists entirely of JavaScript files. We run Fuzzilli to evaluate whether existing JavaScript engine fuzzers can discover vulnerabilities by mutating Mad-Eye's initial corpus. From Figure 2, Fuzzilli achieves relatively high code coverage, ranking second after Mad-Eye. Incorporating Wasm regressions [16–18] demonstrates better Wasm coverage than existing generative Wasm fuzzers during 90 CPU-day fuzzing. In Table 2, Fuzzilli identified five reachable assertions in JavaScriptCore, while Mad-Eye detected these five and 12 additional vulnerabilities. These results confirm that although Fuzzilli can generate Wasm and JavaScript code, it is less effective at producing new PoCs.

5.3 RQ3: Ablation Study

In this subsection, we evaluate the effectiveness of Mad-Eye's components in detecting vulnerabilities, using the same experimental setup described in §5.1.1. We additionally measure code coverage for the ablated tools and Mad-Eye using Clang's source-based coverage [12]. As mentioned, all experiments were conducted on partially instrumented JS engines and use the same initial corpus as in §5.1.

Mad-Eye comprises the following components for ablation: a *wasm-generator*, a constrained Wasm mutator, and a guided JS code generator. The guided JS code generator includes several interdependent techniques, *i.e.*, the probing strategy used for variable-guided and shape-aware JS code generation. Due to their strong interdependency, we do not separately evaluate their impacts. 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 generations of imports and exports interaction interfaces. We then implement four prototypes: Mad-Eye-uncon, which conducts unconstrained Wasm mutation and random JavaScript generation upon *wasm-generator**; Mad-Eye-Jcon, which only enables guided JS code generation upon Mad-Eye-uncon; Mad-Eye-Wcon, which only enables constrained Wasm mutation upon Mad-Eye-uncon; and Mad-Eye, which enables all components.

Vulnerability Discovery. Table 2 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 2); and 2) those triggered in the versions tested by Mad-Eye or the ablated tools, but no longer reproducible in the 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**. Mad-Eye-Jcon performs slightly better than Mad-Eye-uncon by discovering one more vulnerability. Mad-Eye-Wcon identifies seven additional vulnerabilities than Mad-Eye-uncon by leveraging the constrained mutation on Wasm code. Through guided JS code generation, Mad-Eye outperforms Mad-Eye-Wcon by detecting six more vulnerabilities, all related to JavaScript-Wasm interactions.

Coverage. The trends in code block coverage are presented in Figure 4, with the final coverage data detailed in Table 3. *wasm-generator** has the lowest coverage among the three engines. Mad-Eye-uncon and Mad-Eye-Jcon show similar coverage because, without constrained Wasm mutation, the JavaScript mutators can easily invalidate the Wasm module. It results in guided JavaScript generation having little effect since the Wasm module cannot be executed. Mad-Eye and Mad-Eye-Wcon show the highest coverage levels. They have similar coverage because we measured the Wasm-related coverage in the engines, while guided JavaScript generation does not produce new Wasm code. However, JavaScript code generation for JS-Wasm interaction is crucial for revealing vulnerabilities as illustrated in Table 2 and §5.1.3.

We further analyzed the source-based coverage results of the instrumented files. The primary reasons for the unexecuted code are: 1) JIT optimizer (*e.g.*, TurboFan) for Wasm shows relatively low coverage as Mad-Eye is not explicitly designed to target the JIT, and 2) header files contain much code but often show low coverage. Exploring Wasm-JIT interactions is left for future work.

6 Discussion

Prior works [38, 48] analyze cross-language programs (e.g., Rust-C, FFI, JS-native extensions) to identify their memory-safety bugs. Mad-Eye instead tests the underlying execution engine (written in C++) that runs such programs by generating cross-language interactions as fuzzing inputs. In this work, we test V8, SpiderMonkey, and JavaScriptCore, as they are the JavaScript engines embedded in mainstream browsers that execute untrusted JavaScript and Wasm code from potentially malicious websites. Mad-Eye can test other JavaScript engines. For instance, 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 such as JerryScript [9] and Duktape [5], as well as standalone Wasm runtimes like Wasmtime [20], fall outside our scope since they currently lack support for JS-Wasm interactions.

Besides, Wasm modules can interact with each other by importing and exporting objects. These interactions may also introduce security risks. Currently, Mad-Eye generates one Wasm module per test case and does not handle inter-module linkage for interactions, therefore missing potential vulnerabilities in those interactions. Additionally, vulnerabilities unrelated to Wasm features have been extensively tested and are outside the scope of this work.

7 Related Work

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 [41], deep learning [37], reinforcement learning [30], language or graph-based IRs [31, 49], or generative approaches based on grammars [23, 42, 43] or IRs [28, 31]. The industry also actively

Table 3: Coverage data of ablated tools and other works. 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
Ablation	*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%	59.14%	73.82%	60.49%	55.49%	58.72%	70.22%	67.05%	53.77%
	Mad-Eye-Jcon	47.03%	58.43%	52.08%	43.07%	59.16%	73.8%	60.52%	55.56%	59.82%	71.28%	69.0%	55.06%
	Mad-Eye-Wcon	**49.28%**	59.93%	**54.9%**	**46.61%**	65.8%	77.74%	68.02%	65.61%	63.72%	74.52%	72.34%	59.25%
Comparison	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%**

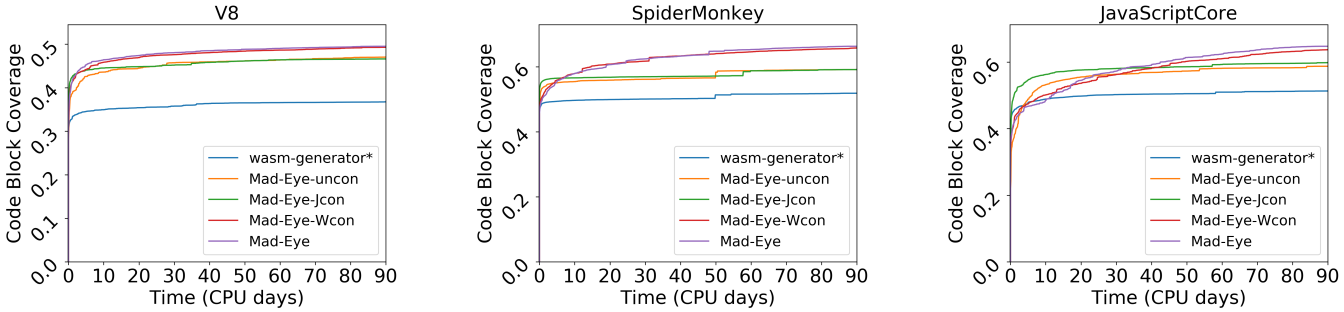

Figure 4: Code block coverage trend of ablated tools and Mad-Eye.

tests JavaScript engines. For example, Google operates Fuzzilli for testing V8 and other engines [7]. Recently, most of these efforts have focused on the JIT component using differential techniques [24, 30, 31, 34, 36, 44, 45], which check output consistency before and after JIT optimizations. Favocado [29] also tests cross-language issues through JavaScript bindings by generating JavaScript programs to invoke interfaces implemented in C++. Prior work has successfully reported vulnerabilities that cause crashes or 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.

Wasm Runtime Security. Some works focus on detecting vulnerabilities or bugs in Wasm runtimes. RGFuzz [40] implements a generative approach to detect crashes and semantic bugs in Wasm runtimes. WASMaker [26] collects and mutates real-world Wasm bytecode to conduct differential testing on Wasm runtimes. Wapplique [50] 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 caused by JS-Wasm interactions. Other works propose methods to protect Wasm runtimes. Johnson *et al.*[35] propose a verifiably sandbox that maintains access isolation for the host OS's storage and network resources. Bosamiya *et al.*[25] 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.

Cross-language Testing. There are vairous research works also focusing on cross-language targets. For example, *Favocado* [29] tests JavaScript binding layers (written in C++) by generating JS code. *PolyFuzz* [38] proposes holistic greybox fuzzing with cross-language coverage for C/Python/Java stacks at the application level. *Atlas* [48] attempts an automated fuzzing framework for closed-source native libraries on Android. It performs cross-language analysis for Java bytecodes and native C/C++ libraries to test the native APIs. Compared to them, which primarily focus on detecting vulnerabilities in code itself written in multiple languages, Mad-Eye targets the *engines* that execute cross-language programs.

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 generate JavaScript or Wasm code independently, Mad-Eye generates both types of code in an interdependent manner. To achieve this, we propose a novel cross-language code fusion technique that generates diverse cross-language interactions to test vulnerabilities in JavaScript engines. Our evaluation across V8, SpiderMonkey, and JavaScriptCore identified 21 previously unknown vulnerabilities, 20 of which have been confirmed, and 18 have already been fixed by browser vendors. Notably, compared to related works, Mad-Eye can achieve higher code coverage on Wasm and discover vulnerabilities that other fuzzers fail to detect.

9 Acknowledgement

We thank the anonymous reviewers for their constructive comments on improving our work. This work is supported by the NSFC for Young Scientists of China (No.62202400) and the RGC for Early Career Scheme (No.27210024). Any opinions, findings, or conclusions expressed in this material are those of the authors and do not necessarily reflect the views of NSFC and RGC.

References

[1] 2017. V8's internal Wasm generator. https://chromium.googlesource.com/v8/v8/+/52d0c2d57527580dbd491e30fc7988384e54fd0c/test/fuzzer/wasm/fuzzer-common.cc.

[2] 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.

[3] 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/.

[4] 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.

[5] 2024. Duktape - embeddable Javascript engine with a focus on portability and compact footprint. https://github.com/svaarala/duktape.

[6] 2024. Emscripten. https://emscripten.org/.

[7] 2024. Fuzzilli in Google. https://github.com/v8/v8/tree/main/test/fuzzilli.

[8] 2024. JavaScript Objects and Shapes. https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Scripting/Object_basics.

[9] 2024. Jerryscript. https://github.com/jerryscript-project/jerryscript.

[10] 2024. Mutations in fuzzing. https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html.

[11] 2024. RGFuzz repository. https://github.com/kaist-hacking/RGFuzz.

[12] 2024. Source based code coverage. https://clang.llvm.org/docs/SourceBasedCodeCoverage.html.

[13] 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/.

[14] 2024. Type confusion in v8 wasm. https://issues.chromium.org/issues/332081797.

[15] 2024. Wasm Feature Extensions. https://webassembly.org/features/.

[16] 2024. Wasm regression tests in JSC. https://github.com/WebKit/WebKit/tree/main/JSTests/wasm.

[17] 2024. Wasm regression tests in SpiderMonkey. https://github.com/mozilla/gecko-dev/tree/master/testing.

[18] 2024. Wasm regression tests in V8. https://chromium.googlesource.com/v8/v8/+/refs/heads/main/test/mjsunit/wasm/.

[19] 2024. Wasmbuilder. https://chromium.googlesource.com/v8/v8/+/refs/heads/main/test/mjsunit/wasm/wasm-module-builder.js.

[20] 2024. Wasmtime. https://docs.wasmtime.dev/.

[21] 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/.

[22] 2024. WebAssembly JavaScript APIs. https://developer.mozilla.org/en-US/docs/WebAssembly/Using_the_JavaScript_API.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. 2021. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases.. In *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA, USA.

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] Wen Li, Jinyang Ruan, Guangbei Yi, Long Cheng, Xiapu Luo, and Haipeng Cai. 2023. {PolyFuzz}: Holistic Greybox Fuzzing of {Multi-Language} Systems. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1379–1396.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT solvers via semantic fusion. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. London, UK.

[47] Dongwei Xiao, Zhibo Liu, and Shuai Wang. 2023. Metamorphic shader fusion for testing graphics shader compilers. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. Melbourne, Australia.

[48] Hao Xiong, Qinming Dai, Rui Chang, Mingran Qiu, Renxiang Wang, Wenbo Shen, and Yajin Zhou. 2024. Atlas: Automating Cross-Language Fuzzing on Android Closed-Source Libraries. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 350–362. doi:10.1145/3650212.3652133

[49] 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.

[50] 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.