Extraction and Mutation at a High Level: Template-Based Fuzzing for JavaScript Engines

WAI KIN WONG*, Hong Kong University of Science and Technology, Hong Kong
DONGWEI XIAO*, Hong Kong University of Science and Technology, Hong Kong
CHEUK TUNG LAI, VX Research Limited, United Kingdom
YITENG PENG, Hong Kong University of Science and Technology, Hong Kong
DAOYUAN WU, Lingnan University, Hong Kong
SHUAI WANG†, Hong Kong University of Science and Technology, Hong Kong

JavaScript (JS) engines implement complex language semantics and optimization strategies to support the dynamic nature of JS, making them difficult to test thoroughly and prone to subtle, security-critical bugs. Existing fuzzers often struggle to generate diverse and valid test cases. They either rely on syntax-level mutations that lack semantic awareness or perform limited, local mutations on concrete code, thus failing to explore deeper, more complex program behaviors. This paper presents TemuJs, a novel fuzzing framework that performs extraction and mutation at a high level, operating on abstract templates derived from real-world JS programs. These templates capture coarse-grained program structures with semantic placeholders, enabling semantics-aware mutations that preserve the high-level intent of the original code while diversifying its behavior. By decoupling mutation from concrete syntax and leveraging a structured intermediate representation for the templates, TemuJs explores a broader and more meaningful space of program behaviors. Evaluated on three major JS engines, namely, V8, SpiderMonkey, and JavaScriptCore, TemuJs discovers 44 bugs and achieves a 10.3% relative increase in edge coverage compared to state-of-the-art fuzzers on average. Our results demonstrate the efficacy of high-level, template-mutation fuzzing in testing JS engines.

CCS Concepts: • **Security and privacy → Browser security**; • **Software and its engineering → Software verification and validation**; **Compilers**.

Additional Key Words and Phrases: JavaScript engine, fuzzing, security, Just-in-time compilation

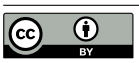
ACM Reference Format:
Wai Kin Wong, Dongwei Xiao, Cheuk Tung Lai, Yiteng Peng, Daoyuan Wu, and Shuai Wang. 2025. Extraction and Mutation at a High Level: Template-Based Fuzzing for JavaScript Engines. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 376 (October 2025), 29 pages. https://doi.org/10.1145/3763154

1 Introduction

JavaScript (JS) enhances user experience by enabling dynamic content updates, animations, and responsive interfaces. It forms the backbone of modern web applications, with about 98.3% of

*Equal contribution.
†Corresponding author.

Authors' Contact Information: Wai Kin Wong, Hong Kong University of Science and Technology, Hong Kong, Hong Kong, wkwongal@cse.ust.hk; Dongwei Xiao, Hong Kong University of Science and Technology, Hong Kong, Hong Kong, dxiaoad@cse.ust.hk; Cheuk Tung Lai, VX Research Limited, London, United Kingdom, darkfloyd@vxrl.hk; Yiteng Peng, Hong Kong University of Science and Technology, Hong Kong, Hong Kong, ypengbp@cse.ust.hk; Daoyuan Wu, Lingnan University, Hong Kong, Hong Kong, daoyuanwu@ln.edu.hk; Shuai Wang, shuaiw@cse.ust.hk, Hong Kong University of Science and Technology, Hong Kong, Hong Kong.



ACM 2475-1421/2025/10-ART376
https://doi.org/10.1145/3763154

all websites using JS [Pawar and Jambhale 2024]. JS features a dynamic typing system and a flexible programming model, which makes it easy to write and is expressive for implementing web functionalities. With over 12 million active developers, JavaScript has one of the largest developer bases [Temple 2024]. Major companies like Google, Facebook, and Microsoft utilize JavaScript extensively, contributing to its widespread adoption and influence [GraffersID 2025]. Moreover, various frameworks and libraries have been developed to facilitate the development of JS-powered web applications, such as React, Angular, and Vue.js.

A JS engine is a complex program that executes JS code, typically found in web browsers. JS engines use a combination of interpreters and compilers to execute code. They employ a multi-tiered Just-In-Time (JIT) compilation strategy and incorporate various optimizations like inlining, loop optimizations, and speculative optimization [Daily 2024]. However, such a design requires complex decision-making to balance quick code execution and optimal performance. Moreover, JS is a dynamically typed language, thereby making it hard for JS engines to infer variable types at compile time. The complexity of JS engines is also evident in their code size, with over a million lines of code in production engines like V8 [Google 2025b].

The correctness of JS engines is crucial for the security and reliability of web applications and the underlying systems. JS engines are a common target for attackers due to their widespread use — over one-third of all attacks involve JavaScript files annually [Pawar and Jambhale 2024]. Vulnerabilities in JS engines, such as type confusion errors (e.g., CVE-2024-5830 [Crawford 2024] in Chrome's V8), can allow attackers to execute arbitrary code on a user's device simply by visiting a malicious website. Hence, it is essential to test JS engines thoroughly to ensure their correctness and reliability.

Fuzzing is a widely adopted technique to test JS engines. Generation-based fuzzing randomly generates JS programs from scratch. While such an approach is fast and flexible in generating test cases, it may be difficult to generate hard-to-model language features, especially given the flexible typing system of JS. Mutation-based fuzzing mutates open-source JS programs to generate new test cases. While the mutation-based approach enjoys the diversity of language features in seed programs, it only performs local mutations to avoid breaking the JS language specifications, thus limiting the diversity of the generated test cases. Even though JS engines are heavily tested by their developers and researchers, new high-severity bugs in these engines are still constantly discovered or even exploited [Crawford 2024; Cyble 2024; TrueFort 2024]. Such observations call for more effective and efficient fuzzing techniques to test JS engines.

To address the limitations of existing JS engine fuzzers, we propose a novel template mutation approach for JS engine fuzzing. We abstract JS programs into templates, which replace concrete expressions with placeholders and preserve the high-level structure of the JS programs. Instead of generating or mutating concrete JS programs, our mutations are performed on the templates. Template mutation allows us to perform more complex and aggressive mutations on the templates due to the reduced constraints of template mutations. The concretized test cases from the mutated templates are thus more diverse and can cover a wider space of the JS engines.

We implement our approach in a fuzzer called TemuJs, and evaluate it on mainstream JS engines, including Google's V8 [Google 2025d], Mozilla's SpiderMonkey [Mozilla 2025c], and Apple's JavaScriptCore [Apple 2025b]. Our evaluation shows that TemuJs finds 44 bugs in the tested JS engines. All of our findings are promptly acknowledged by the developers of the JS engines, and 39 of them are fixed by the time of writing. We also received a bug bounty for one of the bugs we found due to its high severity. Compared with state-of-the-art JS engine fuzzers, TemuJs achieves 10.3% higher edge coverage on average. In summary, our contributions are as follows:

- Conceptually, we introduce a new perspective on fuzzing JS engines. Instead of directly generating and mutating concrete JS programs, we first summarize the characteristics of real-world JS programs into templates. This data-guided fuzzing process at a high template level distinguishes our work from previous JS engine fuzzing approaches, which primarily focus on low-level generation/mutations.
- Technically, we propose a novel pipeline composed of abstraction, mutation, and concretization, featuring innovative placeholder designs, probabilistic sampling, and dataflow metrics. Our approach abstracts JS programs into templates and performs mutations on templates. This method expands the mutation space, enabling more diverse and higher-quality test cases, and effectively overcomes the limitations of existing JS engine fuzzers.
- Experimentally, TemuJs discovers 44 bugs in three mainstream JS engines. It also achieves, on average, 10.3% higher edge coverage compared to state-of-the-art JS engine fuzzers. The bugs identified were promptly fixed by JS engine developers, demonstrating the practical impact of our work in enhancing the security and reliability of JS engines.

2 Background

2.1 JavaScript Language

JavaScript is a high-level and dynamically typed programming language that is widely used in web development. JavaScript is a core technology of the World Wide Web and is supported by all modern web browsers. JavaScript has a set of features that make it distinct from other programming languages. Given the distinct nature of JavaScript, it is important to have a dedicated fuzzer to generate test cases for JavaScript engines. We list some of the key features of JavaScript that make it unique but also challenging to fuzz:

Dynamic Typing. Unlike statically typed languages, such as C or Java, the type of some JavaScript variables cannot be determined until runtime. While this offers developers considerable flexibility, it introduces substantial hurdles for fuzzing JavaScript engines. Generating effective inputs for fuzzing requires intelligent strategies to account for JavaScript's diverse types and uncover vulnerabilities.

Mixture of Object-Oriented and Functional Programming. JavaScript is a multi-paradigm language that supports both object-oriented and functional programming. The ability to define classes and objects alongside higher-order functions and closures allows for complex interactions between these paradigms. Fuzzers struggle to generate test cases that adequately cover the diverse code paths and interactions resulting from this mixture of programming styles.

Asynchronous Programming. JavaScript is single-threaded and uses an event-driven model to handle asynchronous operations. Asynchronous operations in JavaScript are typically handled using callbacks, promises, or async/await. Generating effective test cases requires careful coordination of asynchronous operations and their corresponding callbacks, thus making it challenging for fuzzers to explore the full range of asynchronous behaviors in JavaScript engines.

2.2 JavaScript Engines

JavaScript engines are the interpreters or compilers that execute JavaScript code in web browsers. The execution of a piece of JS code can be in two modes: interpreted mode and compiled mode. In interpreted mode, the engine parses the code and executes it line by line, while in compiled mode, the JS engine employs just-in-time (JIT) compilation to compile the code into machine code and execute the compiled code. The interplay of these two modes of execution, as well as the complexity of the JavaScript language, makes JavaScript engines highly complex and challenging to develop. The complexity of JavaScript engines also makes them prone to bugs, which can lead to security

vulnerabilities in web applications. It is thus imperative to test JavaScript engines to ensure their correctness and security.

2.3 Existing Efforts in Fuzzing JavaScript Engines

Fuzzing JS engines attracts significant attention due to the pervasiveness of JavaScript in web applications and the potential security threats that vulnerabilities in JS engines can bring. Detecting bugs in JS engines is challenging due to the complexity of both the engines themselves and the JS language. Fuzzers for JS engines need to craft test cases that can effectively trigger unknown bugs in the engines. Based on the techniques used to generate test cases, existing fuzzers for JS engines can be roughly categorized into two types: generation-based and mutation-based.

Generation-Based Fuzzers. Generation-based fuzzers generate test cases from scratch. Context-free grammar-based fuzzers like jsfunfuzz [Mozilla 2023] generate test cases by expanding the grammar rules of JavaScript. However, these works have limitations in generating semantically valid test cases. More advanced generation-based fuzzers like Fuzzilli [Groß et al. 2023] design an Intermediate Representation (IR) for representing JavaScript programs and generate test cases by first generating the IR and then translating the IR to JavaScript. FuzzFlow [Xu et al. 2024] creates a graph representation for the JavaScript program to better capture the control and data flow of the program and facilitates the generation of complex test cases with the graph IR. Despite the advances in generation-based fuzzers, they still have difficulty generating test cases that can thoroughly cover the functionalities of JS engines.

Mutation-based Fuzzers. Mutation-based fuzzers mutate existing test cases to generate new test cases. Superion [Wang et al. 2019] ports AFL [Fioraldi et al. 2020] to JS programs by defining the grammar of JS programs and performing mutations according to the grammar. DIE [Park et al. 2020] increases the mutation effectiveness by stochastically preserving certain properties in the program. Some works like Montage [Lee et al. 2020] and CovRL-Fuzz [Eom et al. 2024] employ neural networks to enhance the diversity of the mutated test cases. Nonetheless, their mutations are still limited to the low-level details of JS programs, and they may miss bugs that require inputs with intricate semantics to trigger.

Enhancing Bug Detection Capability with Better Oracles/Coverage Metrics. Instead of focusing on the test case generation process, some works improve the bug detection capability of JS engines by providing stronger testing oracles (ground truth) [Wachter et al. 2025a; Wang et al. 2023] or better feedback mechanisms [Eom et al. 2024; Wang et al. 2024]. However, these approaches are orthogonal to the test case generation process and can be combined with better test case generation schemes to further improve the bug detection capability of JS engines. In this work, we focus on improving the test case generation process to detect vulnerabilities in JS engines. Our work is complementary to these approaches and can be combined with them to further enhance the bug detection capability of JS engines.

3 Motivation

Despite the abundance of existing fuzzers for JS engines, generating test cases that are effective to trigger bugs in the engines is still challenging. In the following, we will introduce several deficiencies in the existing paradigms of test case generation for JS engines. We will also show our approach to addressing these limitations, and use real bugs found by TemuJs for illustration.

Challenge 1: Generation of Certain Language Features is Hard. Generation-based fuzzers are often implemented as a rule-based generator that combines different language features to generate test cases. For instance, some fuzzers like Fuzzilli [Groß et al. 2023] and FuzzFlow [Xu et al. 2024] use IR to generate programs that are syntactically and semantically correct. As mentioned in Section 2.1, JavaScript has a set of features that make it distinct from other programming languages,

```
const v0 = `
  const v4 = Array(v2);
  class C5 {
      static o(a7,a8,a9) {
        v4.__proto__ = v2;
      }
  }`;
%RuntimeEvaluateREPL(v0);
```

(a) Bug-triggering program.

```
// Concrete program
const v0 = `let a = 42;`;
%RuntimeEvaluateREPL(v0);

// Extracted template
const v0 = <code_str>;
%RuntimeEvaluateREPL(v0);
```

(b) The concrete program and the extracted template for triggering the bug.

Fig. 1. A real-world bug found uniquely by TemuJs with template extraction and concretization.

such as dynamic typing and asynchronous programming. Modeling the semantics of these features is challenging, and as a result, many fuzzers fail to generate test cases that can thoroughly cover the functionalities of JS engines.

Solution to Challenge 1: Template Extraction and Concretization. To address the limitations of existing fuzzers, we propose to capture program sketches as templates. A template is composed of holes that need to be filled, and concretizing the template is the process of filling the holes with concrete expressions and statements. A template abstracts away the low-level details of the program while preserving its high-level, coarse-grained structure. Such abstraction enables easier modeling of the semantics of the language features. We design multiple types of placeholders to represent different semantics of the programs and employ these placeholders to generate targeted test cases. Moreover, given the plethora of open-source JS programs, we can easily extract a large number of JS templates from these programs, and concretize the templates for new test cases.

Example. Fig. 1 shows a real bug found by TemuJs but missed by other fuzzers.[1] The dynamic nature of JS language makes it possible to run a string as a program. The `RuntimeEvaluateREPL` function is an intrinsic function that evaluates a string as a JS program in the REPL environment. While the versatility of this function is useful for developers, it also makes the automatic generation of test cases containing this function challenging. Randomly generating a string argument for this function is unlikely to trigger the bug, as the string needs to contain a sophisticated JavaScript class definition. Existing fuzzers often fail to generate such test cases due to the complexity of the language features involved. Instead, we design a `code_str` placeholder (see Fig. 1b) to indicate that the placeholder should be concretized with a valid JS program. The upper and lower portions of Fig. 1b show the concrete program and its extracted template, respectively. The extracted template is then concretized with a sophisticated JS class definition (Fig. 1a), and the resulting test case triggers a bug in V8 from Chrome. This bug was promptly confirmed and fixed by the V8 team, and the corresponding test case was merged into the internal test suite of V8 due to its high quality.

Challenge 2: Limited Mutation Space. Mutation-based fuzzers enjoy the advantage of relying on seed programs with hard-to-model language features to enrich the diversity of their test cases. However, they are often constrained by the limited mutation space. These fuzzers employ human-defined mutation rules to mutate the seed programs, yet the mutation rules are often confined by constraints related to the low-level details of the programs and thereby fail to harness the full potential of seed programs. For instance, to preserve the constraint that a variable should be defined before being used, or to preserve the type validity of expressions, the mutation rules often have to be conservative and perform local mutations to avoid breaking such constraints. Hence,

[1]We simplify the bug-triggering program for readability, as the original program is more complex. The same applies to examples in the rest of the paper.

```
let a = {};
a = -2027262632;
const b = Reflect ?? -0;
function foo() {
  a = b;
  return b;
}
foo();
```

(a) Bug-triggering program.

```
function foo() {
  a = -3;
  return (a ** 69) != -1;
}
foo();
```

(b) Seed program 1.

```
let b = -1;
b = 0;
const c = Reflect ?? 0;
```

(c) Seed program 2.

```
let <var> = <expr>;
<var> = <cnst>;
const <var> = Reflect ?? -0;
function foo() {
  <var> = <expr>;
  return <expr>;
}
foo();
```

(d) Fused template.

```
function foo() {
  <var> = <expr>;
  return <expr>;
}
foo();
```

(e) Seed template 1.

```
let <var> = <expr>;
<var> = <cnst>;
const <var> = Reflect ?? <cnst>;
```

(f) Seed template 2.

Fig. 2. A real bug in V8 found uniquely by TemuJs by fusing two seed templates.

bugs that require more drastic changes to seed programs are less likely to be triggered by existing mutation-based fuzzers.

Solution to Challenge 2: Template Mutation. We propose to mutate templates instead of concrete programs. Compared with mutating concrete programs, mutating templates considerably reduces the constraints on the mutation rules, as only the skeletons of programs are preserved. This allows the mutation to explore a larger space of test cases. Moreover, the mutated templates open up new possibilities to generate test cases with significantly different data flows. Consider fusing two templates with n_1 and n_2 holes, respectively. The fused template with $n_1 + n_2$ holes can be concretized into $2^{n_1+n_2}$ different test cases even if each hole can be filled with only two variables. The fused template thus allows for the generation of diverse data flow patterns that are unlikely to be generated by mutating concrete programs.

Example. Fig. 2 showcases a real bug found by TemuJs in V8 by template mutation. This program is generated with the template in Fig. 2d, which is fused from two seed templates in Fig. 2e and Fig. 2f. This bug can only be triggered under strict conditions: variable `a` initialized with an empty object and then re-assigned in a function with the value of variable `b`, which is a constant initialized with `Reflect ?? -0`.[2] Such complex data flow patterns are not present in either of the seed programs and thus simply fusing or mutating the seed programs is unlikely to trigger this bug. TemuJs, instead, abstracts the seed programs (Fig. 2b and Fig. 2c) into templates (Fig. 2e and Fig. 2f), which eliminate the concrete expressions and values and only preserve the high-level structure of the programs. The templates are then fused into the template in Fig. 2d. The fused template combines program features from the two seed programs. The template, due to the elimination of concrete expressions, does not confine a specific data flow pattern, which allows for drastically different test

[2] `Reflect` [Mozilla 2025b] is a built-in object with methods for interceptable JS operations. The ?? operator [Mozilla 2025a] is the nullish coalescing operator that returns the right-hand operand when the left-hand operand is `null` or `undefined`.

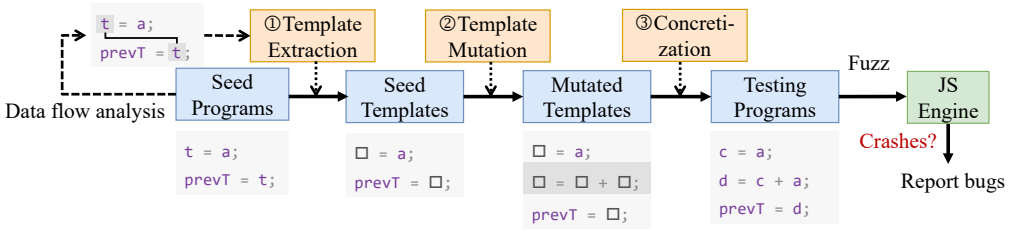


Fig. 3. Testing pipeline of TemuJs.

cases to be generated. The placeholders in the fused template are then filled with expressions from the program generator, leading to the generation of the bug-triggering program in Fig. 2a.

4 Design of TemuJs

Study Scope. TemuJs is designed to detect vulnerabilities, e.g., crashes and memory corruptions, in JavaScript engines. We do not aim to detect logic bugs that lead to incorrect outputs of JS programs since detecting such bugs faces the testing oracle problem, i.e., it is hard to determine the correct output of a JS program. Besides, according to our discussions with JS engine developers, security vulnerabilities are prioritized over logic bugs in JS engines due to the potential security risks.

Fig. 3 illustrates the testing pipeline of TemuJs. TemuJs generates test cases and then feeds the generated test cases into the JS engine under test. The test case generation of TemuJs is composed of three main components — template extraction, template mutation, and concretization.

① **Template Extraction.** Given a set of seed JS programs, the template extraction component extracts templates from the programs. It replaces specific code fragments, such as variables and constants, with placeholders. When deciding the code snippets to replace, the template extraction component analyzes the data flow of the program to identify promising code snippets to replace. We also design specialized placeholders to represent different types of expressions, thus comprehensively abstracting the semantics of the original code. These placeholders are the building blocks of the extracted templates. The extracted templates are stored in a template pool for further mutation.

② **Template Mutation.** This component mutates the templates in the seed pool to generate new templates. Unlike traditional mutation-based fuzzing, which mutates concrete programs, template mutation operates on templates, which abstract away the concrete details of the programs. We thus design mutation operators tailored for templates for effective mutation.

③ **Concretization.** The concretization component fills the holes in JS templates by generating expressions or statements corresponding to the type of the holes. During the generation process, the concretization component maintains the context information for each hole, such as variables in scope and the control flow information, thus ensuring the correctness of the generated programs.

The generated JS programs are fed as testing inputs to the JS engines under test. The JS engines execute the programs. If a crash or memory corruption is detected, we deem the corresponding JS program as an error-triggering test case. We then analyze and report bugs with the collected error-triggering test cases.

In the following sections, we will detail the design of each component of TemuJs. Section 4.1 will introduce the placeholder design in the template extraction process. We design specialized placeholders to represent different semantics of the substituted code fragments. In Section 4.2, we will introduce the substitution metric to decide which code fragments to replace with placeholders. In Section 4.3, we will detail the mutation operators designed for the template mutation process. We will elaborate on the concretization process in Section 4.4.

4.1 Placeholder Design

Design Goal. Template extraction transforms parts of the concrete JS programs into placeholders. Templates are composed of placeholders and concrete code fragments that are not replaced with placeholders. The placeholders are used as abstract representations of the code fragments that they substitute, thus effectively capturing the high-level semantics of the original code and abstracting away concrete details. As such, the design of placeholders directly influences the expressiveness and quality of the extracted templates. Denote the expression to substitute as e and the corresponding placeholder as $\square$. The goal of the placeholder design is to derive a mapping σ that maps e to $\square$, i.e.,

$$\textit{Placeholder Design:} \quad \sigma ::= [e \mapsto \square] \tag{1}$$

Technical Challenge. A naïve way of extracting templates is to replace every constant with a `<cnst>` placeholder and every variable with a `<var>` hole, indicating that a constant/variable should be generated for the corresponding hole during the concretization process. However, such a coarse-grained placeholder design cannot effectively capture the nuances in the semantics of the substituted expressions. The naïve approach only captures the syntactic structure of the replaced expressions, while losing the surrounding context information of the replaced code fragments. For instance, substituting variables with `<var>` placeholders does not distinguish local variables defined in a function from property names of an object. The original structure of the seed programs containing object property names is thus misrepresented in the extracted templates. Some bugs can only be triggered in the presence of such object property names; yet the naïve approach could generate a local variable during the concretization of the `<var>` placeholder, thereby missing the opportunity to trigger the bug.

Multi-Granularity Placeholder Design. To address the misrepresentation issue of the naïve placeholder design, we propose a fine-grained placeholder design that captures the semantics of the replaced code fragments in a more nuanced manner. We design specialized placeholders to represent different types of expressions, thus comprehensively capturing the syntactic/semantic information in the original code. Unlike coarse-grained placeholder designs that create overly broad abstractions and may lose crucial details in the template representation, our method preserves essential structure and relationships within concrete JS programs. By designing fine-grained hole types, such as `<instance_property>` holes as substitutes, we enable the fuzzer to explore a more targeted search space, increasing the likelihood of uncovering subtle JS engine bugs.

Example of Placeholder Types. Take the provided code snippets in Fig. 4 as an example. Fig. 4a shows a concrete JS program containing class definitions `A` and `B`, and Fig. 4b illustrates the extracted template from the concrete JS program in Fig. 4a. Instead of naïvely replacing elements with generic placeholders, we use specialized placeholders that reflect the semantic role of the replaced code. We differentiate between function local variables (`<var>`), loop variables (`<loop_var>`), and class properties (`<instance_property>`, `<static_property>`, etc.). In terms of the class-based structure, we design placeholders to represent static properties (`<static_property>`), instance properties (`<instance_property>`), and built-in properties of objects (`<instance_built_in>`). The placeholders also capture the nuances between literals, such as integers (`<integer>`), integer ranges (`<range(1, 5)>`), and code strings (`<code_str>`). When an expression is an instance of multiple placeholder types, we randomly select one of the corresponding placeholders to replace the expression. For instance, variable `i` in line 8 can be matched to either `<loop_var>` or `<var>`, and we randomly select `<var>` to replace `i`. The similar rule applies to the literal 3 in line 9, in which we randomly substitute the literal 3 with `<range(1, 5)>` instead of `<integer>`.

Definition of Placeholder Types. The placeholders are highly related to the concrete syntax of the JS programs, as placeholders are used as abstract representations of the code fragments they substitute. We show the syntax of an annotated Abstract Syntax Tree (AST) (Λ_{TEMUJS}) in Fig. 5,

```
class A {
  static s = "hello";
  constructor() {
    this.t = 0;
  }
  foo() {
    for (let i = 0; i < 10; i++) {
      this.__proto__ = i; // Built-in
      if (i > 3)
        A.s = "world";
    }
  }
}
class B extends A {
  constructor() {
    super();
    super.t = 1; // Super instance
  }
  foo() {
    super.foo(); // Inherited
    eval("a = 3");
  }
}
```

(a) Concrete JS program.

```
class A {
  <static_property> = <string>;
  constructor() {
    <instance_property> = <integer>;
  }
  t() {
    for (let i = 0; i < 10; i++) {
      <instance_built_in> = <var>;
      if (<loop_var> <cmp> <range(1, 5)>)
        <static_property> = <string>;
    }
  }
}
class B extends A {
  constructor() {
    super();
    <super_property> = <integer>;
  }
  foo() {
    <super_method>();
    eval(<code_str>);
  }
}
```

(b) Placeholders corresponding to the concrete program.

Fig. 4. An example of placeholder design in TemuJs.

ObjectName	o	::=	`this` | *valid object names*
Variable	v	::=	*valid variable names*
ClassName	c	::=	*valid class names*
InstanceProperty	ip	::=	$o.v$
SuperProperty	$supp$	::=	`super`.v
StaticProperty	stp	::=	$c.v$
MethodName	m	::=	*valid method names*
InstanceMethod	im	::=	$o.m$
StaticMethod	stm	::=	$c.m$
Range	r	::=	$\{low, low+1, \ldots, high\}$
UnaryOp	$\diamond$	::=	`not` | `negate`
BinaryOp	$\otimes$	::=	$+ \mid - \mid \times \mid > \mid < \ldots$
Integer	int	::=	$\{0, -1, 1, -2, 2, \ldots\}$
String	str	::=	*valid string literals*
Expression	e	::=	$v \mid e_1 \otimes e_2 \mid \diamond e \mid int \mid \ldots$
Statement	s	::=	$v = e \mid$ if e then s else $s \mid \ldots$
Placeholder	$\square$	::=	$[\![r]\!] \mid [\![int]\!] \mid [\![e]\!] \mid [\![s]\!] \mid \ldots$

Fig. 5. Selected syntax of the Λ_{TemuJs}.

which we use to represent concrete JS programs and derive placeholder types accordingly. The AST defines various JS program features, such as instance/static properties, binary/unary expressions, and literals. The placeholder types are bound to the syntax of the AST, and we define holes $\square$ that correspond to the different constructs in the AST. We define structural placeholders, including $[\![\mathit{InstanceProperty}]\!]$ (for substituting instance properties such as `this.t` in Fig. 4) and $[\![\mathit{SuperProperty}]\!]$ (for substituting property accesses of the `super` object, such as `super.t` in Fig. 4). These structural placeholders preserve object-oriented semantics. We also have value-level placeholders, such as $[\![\mathit{Integer}]\!]$ and $[\![\mathit{Range}]\!]$. The placeholder $[\![\mathit{Range}]\!]$ can be used to replace any integer literal that falls within the corresponding range. The range span is heuristically set to be $\pm \min(2, 0.1v)$ for the original value v, based on empirical tests balancing variation and validity (e.g., avoiding out-of-memory errors or excessively long loops).

Substitution of Concrete Code with Placeholders. The substitution of expressions with placeholders is performed on the parsed AST of the concrete JS programs. The parsed AST is a tree structure. We show the algorithm of replacing concrete expressions with placeholders in Algorithm 1. We initialize *hole_set*, which records all the placeholder types that can be used to replace the AST node, as an empty set $\emptyset$. We then iterate over all AST construct types and check if the AST node is an instance of each type. The InstanceOf function is performed by a simple type checking on the AST node, which can be implemented with the help of JS parsers like Esprima [Hidayat 2025]. For example, an integer literal can be both *Integer* and *Range*. If so, we add the corresponding placeholder $[\![\tau]\!]$ to *hole_set*. Finally, we randomly sample a hole type from *hole_set* and replace the AST node with it.

Algorithm 1 Concrete expression substitution with placeholders.

```
1: function SubsConcrete(node: AST node to be substituted, T: AST tree)
2:     hole_set ← ∅
3:     for τ ∈ {Integer, Range, ...} do                ▷ Iterate over all placeholder types
4:         if InstanceOf(node, τ) then                 ▷ Check if the node is an instance of the AST type
5:             hole_set ← hole_set ∪ [[τ]]
6:     □ ← Sample(hole_set)
7:     Replace(node, T, □)                             ▷ Replace the node with the sampled hole
```

4.2 Data Flow-Aware Template Extraction

Design Goal. Given a concrete expression, the template extraction component can use the method in Section 4.1 to replace the expression with a placeholder. However, it is crucial to decide which expressions to replace with placeholders. The design goal of selective template extraction is to design a probability distribution $\mathcal{D}(P, e)$ to decide the probability of replacing an expression e in the program P. The probability of substituting an expression with a placeholder, i.e., $\mathbb{P}([e \mapsto \square])$, conforms to the distribution:

$$\mathbb{P}([e \mapsto \square]) \simeq \mathcal{D}(P, e) \tag{2}$$

Technical Challenge. Not all data flow paths in a concrete JS program contribute equally to the bug-finding capability of the generated test cases. Some paths are more likely to trigger bugs in JS engines, and replacing code fragments in these paths with placeholders can lead to the generation of more effective test cases. For instance, in a loop that iterates over an array, the array index is more likely to be used in array accesses, and replacing the array index with a placeholder can lead to the generation of test cases that stress-test the array access functionalities of JS engines. On the other hand, replacing all expressions with placeholders may not be effective, as spending resources

on mutating less important code fragments is less likely to lead to the discovery of bugs and wastes resources. We thus need a metric to select code fragments to replace with placeholders.

Data Flow-Aware Substitution Metric. We design a data flow-aware metric to decide which code fragments to replace with placeholders. Intuitively, variables with more complex data flow dependencies are more likely to be the crux of the error-triggering conditions. This is because JIT compilation in JS engines often finds complex data dependencies more challenging than simple ones, making them more prone to bugs. We thus design a metric that assigns a score to each variable based on its data flow complexity.

Definition 4.1. (Data Flow Complexity): Denote the number of define statements that write to a variable v as $\textsc{DefCount}(v)$ and the number of use statements that read from v as $\textsc{UseCount}(v)$. We define the data flow complexity of v as the sum of the number of define and use statements that involve v, i.e., $\textsc{DFComp}(v) = \textsc{DefCount}(v) + \textsc{UseCount}(v)$.

Example. In the code snippet of Fig. 4, the variable `i` in line 8 has a data flow complexity of 6, as it is involved in 6 different define/use statements — the initialization, the loop condition, the read and write operation in the iterative increment, the assignment in line 8, and the comparison in line 9.

Definition 4.2. (Substitution Probability): Given a statement s in a JS program, let $\textsc{UseSet}[s]$ be the set of variables used in s and $\textsc{DefSet}[s]$ be the set of variables defined in s. The data flow complexity of the statement s is calculated as

$$stmt\_df\_comp = \sum_{v \in \textsc{UseSet}[s] \cup \textsc{DefSet}[s]} \textsc{DFComp}[v]. \tag{3}$$

, meaning that the data flow complexity of a statement is the sum of the data flow complexities of the variables involved in the statement. The probability of substitution for the statement s is calculated as the ratio of the data flow complexity of the statement to the total data flow complexity of the program, i.e.,

$$total\_comp = \sum_{v \in V} \textsc{DFComp}[v] \tag{4}$$

$$p = \frac{stmt\_df\_comp}{total\_comp} \tag{5}$$

Selective Substitution. Algorithm 2 shows the selective substitution algorithm based on the data flow complexity metric. The algorithm first calculates the data flow complexity of each variable in the JS program according to the number of define and use statements that involve the variable (lines 2–15). We parse the JS program into an AST T (line 17) and iterate over all the statements in the program (line 18). For each statement, we assign a substitution probability p to each statement according to Eq. (3) and Eq. (4) (line 19). The probability of substitution is calculated by normalizing with the total data flow complexity $total\_comp$, which is computed in line 16. We then recursively iterate over all the descendant nodes of the statement in the AST with depth-first search (line 20). For each child node, we sample a random number r from $[0, 1]$ and replace the node with a placeholder if $r < p$ (lines 21–23). The substitution process is performed by the $\textsc{SubsConcrete}$ function, which is detailed in Algorithm 1.

4.3 Template Mutation

Design Goal. The template mutation component aims to enhance the diversity of templates extracted from seed programs. The component generates new templates by performing mutations on collected templates in the seed template pool. Unlike traditional mutation-based fuzzing, template mutation mutates high-level templates, and thus suffers from fewer constraints in its mutation rules.

Algorithm 2 Data flow-aware selective substitution.

```
1: function SubsDataFlow(P: JS program)
2:     DefCount ← {}, UseCount ← {}                ▷ Initialize mappings from variables to def/use counts
3:     DefSet ← {}, UseSet ← {}                    ▷ Initialize mappings from statements to defined/used variables
4:     for s ∈ Stmts(P) do
5:         V ← Vars(s)                             ▷ Get the variables in the statement
6:         DefSet[s] ← DefVars(s)                  ▷ Get the variables defined in the statement
7:         UseSet[s] ← UseVars(s)                  ▷ Get the variables used in the statement
8:         for v ∈ V do
9:             if v ∈ DefSet[s] then
10:                DefCount[v] ← DefCount[v] + 1   ▷ Increment the define count
11:            if v ∈ UseSet[s] then
12:                UseCount[v] ← UseCount[v] + 1   ▷ Increment the use count
13:    DFComp ← {}                                 ▷ Mapping from variables to data flow complexity
14:    for v ∈ Vars(P) do
15:        DFComp[v] ← DefCount[v] + UseCount[v]   ▷ Calculate the data flow complexity
16:    total_comp ← ∑_{v∈V} DFComp[v]              ▷ Calculate the total data flow complexity
17:    T ← Parse(P)                                ▷ Parse the JS program into an AST
18:    for s ∈ Stmts(P) do
19:        p ← ∑_{v∈UseSet[s]∪DefSet[s]} DFComp[v]/total_comp   ▷ Calculate the probability of substitution
20:        for node ∈ Dfs(s) do                    ▷ Depth-first search the AST
21:            r ← Sample([0, 1])                  ▷ Sample a random number
22:            if r < p then
23:                SubsConcrete(node, T)           ▷ Replace the node with placeholders
```

Given a template P extracted from a seed program, the goal of template mutation is to transform P through a series of mutation operators $o_1, o_2, \ldots$, into a new template P', i.e.,

$$\textit{Template Mutation:} \quad P \xrightarrow{o_1} P_1 \xrightarrow{o_2} P_2 \xrightarrow{\cdots} P' \tag{6}$$

Technical Challenge. Due to the differences between concrete programs and templates, the mutations for concrete programs may not be directly applicable to templates. Concrete programs are subject to low-level constraints on concrete code structures, and the mutation operators for concrete programs must respect these constraints to avoid generating trivially invalid code. For example, splicing two concrete programs requires analyzing the data flow of the concrete programs to ensure that the variables in the spliced code are in scope and type-correct. In contrast, templates abstract away concrete details, and the mutation operators for templates can operate on incomplete programs with placeholders. The high-level abstraction of templates enables their mutation operators to be more flexible, allowing for more aggressive and semantic-aware transformations.

Template Mutation Operators. Inspired by the mutation operators for concrete programs, we design five types of mutations for templates: ① insertion, ② deletion, ③ substitution, ④ fusion, and ⑤ splicing. Instead of mutating concrete programs, the mutations are performed on placeholders, which allows for more powerful and semantic-aware transformations. The algorithm of the template mutation operators is shown in Algorithm 3. To facilitate the insertion and substitution mutations, we introduce a helper function CreateNewHole that generates a new hole tree to append to/replace the original hole. The function samples a random AST type from the set of valid AST types and checks if the AST type is a terminal node (line 6). A terminal node is a leaf node in the AST, such as literals and variables. The function then creates a new hole tree by adding the new hole to the hole tree until a terminal node is reached (lines 4–8). The function returns the new hole tree to the caller (line 9).

Algorithm 3 Template mutation operators.

```
1: function CreateNewHole
2:     is_terminal ← False
3:     T□ ← ∅                                          ▷ Initialize a tree to store the new holes
4:     while !is_terminal do
5:         τ ← Sample(valid AST types)
6:         is_terminal ← IsTerminal(τ)                 ▷ Check if the AST type is a leaf node type
7:         □ ← ⟦τ⟧
8:         T□ ← Add(T□, □)                             ▷ Add the new hole to the hole tree
9:     return T□
10: function Insertion(T: Parsed AST of the template to mutate)
11:     □ ← Sample(Holes(T))                           ▷ Randomly select a hole in the AST tree
12:     T□ ← CreateNewHole()
13:     Add(T, □, T□)
14: function Deletion(T: Parsed AST of the template to mutate)
15:     □ ← Sample(Holes(T))
16:     Remove(T, □)                                   ▷ Remove the hole from the AST tree
17: function Substitution(T: Parsed AST of the template to mutate)
18:     □ ← Sample(Holes(T))
19:     T□ ← CreateNewHole()
20:     Replace(T, □, T□)                              ▷ Replace the hole with the new hole
21: function Fusion(T1, T2: Parsed AST of the templates to mutate)
22:     □ ← Sample(Holes(T1))                          ▷ Randomly select a hole in the first AST tree
23:     Tf ← Add(T1, □, T2)                            ▷ Add the second AST tree to the first AST tree at the hole
24:     return Tf
25: function Splicing(T1, T2: Parsed AST of the templates to mutate)
26:     T1^s ← Sample(Subtrees(T1))                    ▷ Randomly select a subtree
27:     T2^s ← Sample(Subtrees(T2))
28:     return Fusion(T1^s, T2^s)                      ▷ Fuse the two subtrees
```

① **Insertion.** The insertion mutation inserts new expression holes into a random location in the template. This mutation allows for the generation of more complex templates by adding new control and data flow paths, thus effectively enabling more diverse test cases that cannot be generated by the original templates. For instance, we can generate expression holes $[\![\otimes]\!][\![v]\!]$, and append them to the end of a randomly selected expression to form a binary operation expression $e[\![\otimes]\!][\![v]\!]$. Due to the versatility of the placeholders, the insertion location can be anywhere in the template, thus allowing for more diverse test cases.

② **Deletion.** The deletion mutation removes holes from the template. The mutation is useful for obtaining unexpected test cases by removing certain control/data flow paths. We parse the template into an IR tree that comprises both placeholders and unsubstituted concrete expressions. We then randomly select a placeholder in the IR tree and remove it from the tree (lines 15–16).

③ **Substitution.** The substitution mutation randomly replaces a hole in the template with newly generated expression holes (lines 18–20). For instance, if the hole is a binary operation hole $[\![e_1]\!][\![\otimes]\!][\![e_2]\!]$, we can replace it with a unary operation hole $[\![\diamond]\!][\![e]\!]$. The substitution mutation is useful for generating test cases that involve different types of expressions, thus enhancing the diversity of the generated test cases.

④ **Fusion.** The fusion mutation combines two templates into a single template (lines 22–24). Such mutations enable the generation of test cases that involve multiple control/data flow paths from different templates during the concretization phase. Unlike concrete program fusion, we do not need

to perform variable renaming or substitution to merge data flow paths from different templates, as the variables are abstracted away by the placeholders.

⑤ **Splicing.** The splicing mutation combines two templates by splicing them at a random point (lines 26–28). Instead of fusing the two templates entirely, we randomly select a subtree from each template and fuse them together. This mutation is useful for generating novel data flow patterns from the partial templates, thus enhancing the diversity of the generated test cases.

4.4 Concretization

Design Goal. The concretization component fills the holes in the templates with concrete expressions to generate JS programs that can be executed in JS engines. The process has two primary, measurable goals: validity and diversity. Validity refers to generating syntactically and semantically correct programs that pass the JS engine's parser, measured by the *valid rate* (the percentage of generated programs that are executable). Diversity refers to producing a wide range of programs that exercise different execution paths, measured by the code *coverage* achieved on the target engine. Given a template P, the goal is to perform a series of substitutions to form a concrete program P^c that maximizes both metrics:

$$\textit{Concretization:} \quad P \xrightarrow{[\square_1 \mapsto e_1]} P_1^c \xrightarrow{[\square_2 \mapsto e_2]} P_2^c \xrightarrow{\cdots} P^c, \quad \textsc{Holes}(P^c) = \emptyset \tag{7}$$

Technical Challenge. Concretization suffers from more constraints than template mutation, as the former must conform to the low-level details of JS programs, and the quality of the generated JS programs directly affects the effectiveness of the generated test cases. The concretization faces two main challenges: ① validity, and ② diversity.

① **Validity of Concretized JS Programs.** Invalid JS programs are readily rejected by JS engine parsers, potentially masking deep-seated bugs within those engines. We note that mutated templates are not executable without being concretized. A naïve, context-blind approach would yield a valid rate below 10%, with programs being rejected by the parser and thus failing to exercise deeper engine logic. Unlike high-level templates, in which placeholders within the templates can almost be arbitrarily replaced with other placeholders, the concretized JS programs should respect the syntax and semantics specifications of the JS language. For instance, referencing a non-existent object property or calling an undefined function can lead to a syntax error in JS engines, thus rendering the deep logic of interpreting/compiling objects and functions untested.

② **Diversity of Concretized JS Programs.** The concretization process should generate programs that exhibit different control/data flow paths, thus leading to more comprehensive testing. The diversity of the concretized programs is crucial for the effectiveness of the generated test cases. For example, imagine a template created by combining two separate templates, each possessing distinct data flow paths. Initially, these data flow paths might be disjoint, meaning they don't share any variables. The concretization process should then intelligently fill the holes within this combined template, strategically using variables originating from both of the original templates. This creates a concrete JS program where the previously separate data flow paths are now joined and interconnected, leading to more comprehensive testing.

To achieve high validity and diversity, our concretization algorithm leverages a set of coordinated components. The context analysis module builds a symbol table of available functions and objects, which the expression generator uses to fill holes with valid and type-consistent code.

Context Analysis. Context is a data structure that maintains the state of the program being generated at any given point. Formally, the context C can be defined as a tuple:

$$C = (S_{cls}, S_{obj}, S_{func}, S_{ctl}) \tag{8}$$

Algorithm 4 Context update algorithm.

```
1:  function UpdateClassContext(s: Statement to analyze, C: Context information)
2:      C.S_cls.in_cls ← True                                  ▷ The statement is inside a class definition
3:      C.S_cls.cls_name ← ClassName(s)                        ▷ Get the class name
4:      C.S_cls.cls_names ← C.S_cls.cls_names ∪ {ClassName(s)} ▷ Add class name to available classes
5:      cls_props ← {}                                         ▷ Initialize class properties
6:      cls_props.st_props ← StaticProps(s)
7:      cls_props.ins_props ← InstanceProps(s)
8:      cls_props.st_methods ← StaticMethods(s)
9:      cls_props.ins_methods ← InstanceMethods(s)
10:     if HasSuperClass(s) then
11:         cls_props.sup_cls ← SuperClassName(s)              ▷ Get the superclass name
12:     C.S_cls.cls_props[ClassName(s)] ← cls_props            ▷ Store class properties in the context
13: function UpdateContext(s: Statement to analyze, C: Context information)
14:     if InstanceOf(s, class def) then
15:         UpdateClassContext(s, C)                           ▷ Update context for class definitions
16:     else if InstanceOf(s, loop) then
17:         C.S_ctl.in_loop ← True                             ▷ The statement is inside a loop
18:         C.S_ctl.loop_vars ← C.S_ctl.loop_vars ∪ LoopVars(s) ▷ Add the loop variables to the context
19:         C.S_ctl.loop_depth ← C.S_ctl.loop_depth + 1        ▷ Increment the loop depth
20:     else if InstanceOf(s, obj new) then
21:         cls_name ← NewClass(s)                             ▷ Get the class name of the new object
22:         obj_name ← NewObjName(s)                           ▷ Get the object name
23:         C.S_obj.objs ← C.S_obj.objs ∪ (cls_name, obj_name) ▷ Add the new object to the context
24:     else if InstanceOf(s, func def) then
25:         C.S_func.funcs ← C.S_func.funcs ∪ {(FuncName(s), Params(s))} ▷ Add function info
26:         C.S_func.func_names ← C.S_func.func_names ∪ {FuncName(s)}    ▷ Add function name
```

where S_{cls} contains information about defined classes, including their names, properties (instance and static), methods, and inheritance hierarchy. S_{obj} is the set of instantiated objects and their corresponding classes. S_{func} stores details of defined functions, such as their names and parameters. S_{ctl} tracks control flow information, including loop variables and nesting depth. This context acts as a dynamic symbol table that evolves as the program is generated and analyzed. By querying this context, the concretizer can generate code that is syntactically valid and semantically consistent with the surrounding code, e.g., only accessing properties of existing objects or calling functions that have been defined in the current scope.

Context Update Algorithm. The derivation of context information is shown in Algorithm 4. The function UpdateContext analyzes the context of a statement and updates the context information C accordingly. Specifically, when encountering a class definition, it invokes UpdateClassContext to record details such as the class name, its properties (static and instance), methods, and superclass information (lines 1–12). For other constructs, it updates the context to reflect the current scope. For instance, it tracks whether the current statement is inside a loop and records loop-specific variables (lines 17–19). When a new object is instantiated, it adds the object's name and its class to the context (lines 21–23). Similarly, for function definitions, it stores the function name and its parameters (lines 25–26). For brevity, we omit the context update for structures like if statements and primitive-type objects due to the page limit.

Expression Generator. The expression generator is responsible for generating concrete expressions to substitute holes in the template. For brevity, we show representative expression generation rules in Algorithm 5. The generator leverages the context information C to produce valid and contextually-aware code fragments. For instance, when filling a hole of type $[\![Range]\!]$, it samples an integer from the specified range (line 3). For more complex, context-dependent holes, it consults the

context C. To generate a $[\![SuperProperty]\!]$ access, it retrieves the superclass name from the context, looks up its properties, and samples a valid property to create an expression like `super.v` (lines 5–8). Similarly, for an $[\![InstanceProperty]\!]$, it randomly selects an existing object (or `this` if inside a class) and one of its properties from the context to form a valid property access (lines 10–16). For method calls, such as $[\![StaticMethod]\!]$, it not only selects a valid class and method but also generates appropriate arguments for the call, ensuring semantic correctness (lines 18–21).

Algorithm 5 Expression generation.

```
1: function ExprGen(□: Hole to fill, C: Context information)
2:     if InstanceOf(□, [[Range]]) then
3:         e ← Sample(Range.low, Range.high)                    ▷ Sample an integer from the range
4:     else if InstanceOf(□, [[SuperProperty]]) then
5:         sup_cls ← C.S_cls.cls_props[C.S_cls.cls_name].sup_cls    ▷ Get the superclass name
6:         sup_props ← C.S_cls.cls_props[sup_cls]               ▷ Get the superclass properties
7:         v ← Sample(sup_props.ins_props ∪ sup_props.st_props)
8:         e ← super.v                                          ▷ Generate a super property expression
9:     else if InstanceOf(□, [[InstanceProperty]]) then
10:        if C.S_cls.in_cls then
11:            cls_name, obj_name ← Sample(C.S_obj.objs ∪ (C.S_cls.cls_name, this))   ▷ Sample an object name or the current object, referred to as "this"
12:        else
13:            cls_name, obj_name ← Sample(C.S_obj.objs)
14:        cls_props ← C.S_cls.cls_props[cls_name]              ▷ Get the class properties
15:        v ← Sample(cls_props.ins_props ∪ cls_props.st_props)
16:        e ← obj_name.v                                       ▷ Generate an instance property expression
17:    else if InstanceOf(□, [[StaticMethod]]) then
18:        c ← Sample(C.S_cls.cls_names)                        ▷ Sample a class name
19:        m ← Sample(C.S_cls.cls_props[c].st_methods)          ▷ Sample a static method name
20:        args ← GenArgs(m, C)                                 ▷ Generate appropriate arguments
21:        e ← c.m(args)                                        ▷ Generate a static method call expression
22:    return e
```

Overall Concretization Process. The concretization process is shown in Algorithm 6. The algorithm initializes the context information C (line 2) and iterates over all the statements in the template P (line 3). Before processing the holes within a statement, it first updates the context based on the statement itself (line 4). For each hole, it generates a concrete expression using the current context (line 6) and replaces the hole with this new expression (line 7). The context is then updated again to reflect any new variables, functions, or objects introduced by the concretized expression, ensuring this information is available for subsequent holes (line 8). This process continues until all holes are filled, resulting in a complete and executable JS program (line 9).

Algorithm 6 Concretization process.

```
1: function Concretize(P: Template to concretize)
2:     C ← InitContext()                                        ▷ Initialize context with empty sets and false flags
3:     for s ∈ P do
4:         C ← UpdateContext(s, C)                              ▷ Update context based on the current statement
5:         for □ ∈ Holes(s) do
6:             e ← ExprGen(□, C)                                ▷ Generate a concrete expression for the hole
7:             Replace(s, □, e)                                 ▷ Fill the hole with the concrete expression
8:             C ← UpdateContext(s, C)                          ▷ Update context after filling the hole
9:     return P                                                 ▷ Return the concretized program
```

Table 1. Overview of the evaluated JS engines and the time and commit at the start of the evaluation.

JS Engine	Browser	Commit	Time	Lines of Code
V8	Chrome	e3cdab	Dec. 2024	1.26M
SpiderMonkey	Firefox	e46f83	Dec. 2024	949k
JavaScriptCore	Safari	f018a2	Dec. 2024	629k

5 Implementation of TemuJs

We implement TemuJs in Swift [Apple 2025d], a high-performance and general-purpose programming language developed by Apple. We choose Swift for its performance, safety, and expressiveness. We also use Python to process and analyze the results of the experiments. The execution of the test cases is parallelized using the Grand Central Dispatch (GCD) framework [Apple 2025a] in Swift.

In order to detect any memory corruption bugs, we compile the JS engines with AddressSanitizer [Serebryany et al. 2012] enabled. We also compile the engines with assertions enabled to catch unexpected behaviors during the runtime of the engines, since assertion failures indicate violations of the developers' assumptions about the code and could potentially lead to security vulnerabilities. Following other fuzzing works for JS engines [Groß et al. 2023; Wang et al. 2024, 2023], we enable coverage feedback in the JS engines to guide the fuzzing process. We use the LLVM compiler infrastructure [Lattner and Adve 2004] to compile JS engines with necessary sanitizers and coverage feedback enabled. We use coverage feedback to decide which concrete seed programs to extract templates from; the higher the coverage, the more frequently the concrete seed program is selected. For each selected program in a fuzzing iteration, we extract one template from it.

6 Evaluation

To evaluate the effectiveness and efficiency of TemuJs in uncovering real-world bugs in JavaScript engines, we conduct a comprehensive evaluation and a large-scale fuzzing campaign. Our evaluation aims to answer the following three research questions:

- **RQ1:** Can TemuJs find real-world bugs in JavaScript engines?
- **RQ2:** How does TemuJs compare to the state-of-the-art JavaScript engine fuzzers in terms of code coverage and bug finding ability?
- **RQ3:** How do individual design components of TemuJs contribute to its bug detection capability?

6.1 Experimental Setup

Evaluation Targets. We evaluate TemuJs on three JS engines: V8 from Google, SpiderMonkey from Mozilla, and JavaScriptCore from Apple. We show the details of those engines in Table 1. Our evaluated engines are widely used in practice and have a large user base. Those JS engines are the core components of the Chrome, Firefox, and Safari browsers, respectively. The total share of these three browsers is around 88% of the global browser market [StatCounter 2024]. All three JS engines are open-source and have been under regular and thorough internal testing by their respective companies. We believe that finding bugs in these engines is challenging and meaningful.

Large-Scale Fuzzing Campaign. We evaluate the bug-finding capability of TemuJs by launching a large-scale fuzzing campaign on the three tested JS engines. To unleash the full potential of TemuJs in finding unknown bugs, we run TemuJs on the three testing targets for three months. Our testing campaign is launched in an iterative manner — we regularly update the evaluated JS engines to their latest release version to ensure that we do not trigger old bugs that have been fixed in the latest versions; we also update our fuzzer itself to leverage the latest features and improvements we have developed. The starting time and the commit numbers of the JS engines at the beginning of the evaluation are shown in Table 1.

Experimental Environment. All experiments are conducted on a Ryzen 3970X 32-core server with 256GB of memory. The server is equipped with the operating system of Ubuntu 22.04 LTS, and has 2TB of SSD storage. For JS engine fuzzers that require AFL instrumentation, we use LLVM 20, which is the latest supported version by our evaluated JS engines.

Seed Program Collection. The template extraction of TemuJs extracts templates from a corpus of concrete JS programs. As such, we need to collect a set of seed programs to start the fuzzing process. We adopt similar seed collection approaches to prior JS engine fuzzing works that also require a corpus of test cases for fuzzing [Han et al. 2019; Park et al. 2020]. We collect regression test cases from the bug reports of the evaluated JS engines and the regression test cases used in the CI/CD pipeline of the JS engines [Apple 2025c; Google 2025e; Mozilla 2025d]. After the collection, we obtain 6,803 test cases for each JS engine on average. As a standard practice in fuzzing [Herrera et al. 2021; Wong et al. 2025a], we apply corpus minimization using `afl-cmin` [Zalewski 2025] to reduce the number of test cases for improved efficiency. After the minimization, we get on average 2,505 test cases for each JS engine. We then extract one template from each seed JS program using the techniques in Section 4.1 and Section 4.2.

6.2 RQ1: Bug Finding Capability of TemuJs

Throughput. We run TemuJs for 24 hours and evaluate the throughput of TemuJs in terms of the number of test cases generated and executed. On average, TemuJs produces 488,168, 662,646, and 581,209 test cases per hour on V8, SpiderMonkey, and JavaScriptCore, respectively. The high throughput of TemuJs is attributed to the lightweight nature of the template-based fuzzing approach — we do not employ heavyweight program analysis or symbolic execution, thus allowing for a high throughput of test case generation. By generating hundreds of thousands of test cases each hour, TemuJs is able to efficiently stress-test the JS engines.

Validity of Mutated Templates. We consider a mutated template valid if it can be concretized into syntactically correct JS programs. We find that less than 5% of the mutated templates are invalid. This low rate of invalidity is due to the less constrained nature of template mutations — the template mutations are performed on high-level abstractions, thus effectively reducing the constraints imposed by concrete syntax.

Overall Effectiveness. During the large-scale fuzzing campaign, TemuJs has found 44 bugs in total. Developers are responsive to our findings, and promptly confirmed all of our findings and fixed 39 of them. Due to the page limit, we host detailed information of the bugs found by TemuJs in [Wong et al. 2025b]. Out of the 44 bugs uncovered by TemuJs, 22 bugs have security implications, i.e., they could be used as security exploits to compromise the security of the browsers or even the user systems. Such a high ratio of security bugs illustrates the efficacy of TemuJs in hunting for security-critical bugs in JS engines. In particular, the V8 security team issued a bug bounty for one of the bugs we reported due to its significant security implications. Moreover, two of the bugs found by TemuJs were used as challenges in the CTF competitions and were successfully exploited by white-hat hackers [Google 2025a,c] as their winning solutions. Such observations further demonstrate the real-world impact of the bugs found by TemuJs.

Bug Distribution Among Different JS Engines. We show the bug distribution among JS engines on our website [Wong et al. 2025b]. TemuJs finds 26 bugs in V8. The significant number of bug findings in V8 is highly encouraging, as V8 is the most widely used JS engine in the world, and its corresponding Chrome browser has two-thirds of the global browser market share [StatCounter 2024]. TemuJs also finds 15 bugs in Apple's JavaScriptCore, and its corresponding Safari browser is the second most popular browser in the world and shares 18% of the global browser market [StatCounter 2024]. We also find 3 bugs in SpiderMonkey, which powers the Firefox browser. The bug

distribution differences among the three engines could possibly be due to the different functionalities and design complexities of the engines — as illustrated in Table 1, those three engines have different code sizes. Our findings demonstrate that TemuJs can effectively and consistently find bugs in different JS engines with diverse design and implementation details. We will further discuss the potential of extending TemuJs to test other JS engines in Section 7.

Bug Distribution Among Engine Components. For the fixed bugs found by TemuJs, we analyze the bug patches from the JS engine repositories to identify the erroneous components in the JS engines. We categorize the bugs based on the components in the JS engines that are responsible for the bugs. We find that the JIT compiler is the most error-prone component in the JS engines, responsible for 37 of all the 44 bugs found by TemuJs. The JIT compiler needs to handle complex optimizations and transformations of JS programs, which could lead to subtle bugs that are not present in the interpreter. Existing works have also shown that the JIT compiler is a common source of bugs in JS engines due to its complexity [Bernhard et al. 2022; Groß et al. 2023; Wang et al. 2023]. The parsing and debugging components are less common, responsible for 2 and 1 bugs, respectively. The parsing and debugging bugs could be easier to discover by byte-level fuzzers [Fioraldi et al. 2020; Google 2016], and these components are probably already well-tested by existing fuzzers. Nonetheless, TemuJs is still able to find bugs in these components, demonstrating its capability to uncover new bugs in components that are thoroughly tested by existing techniques. Overall, TemuJs is able to find bugs in diverse components in JS engines, demonstrating its effectiveness in exploring behaviors of the engines.

Answer to RQ1: TemuJs finds 44 bugs in three mainstream JS engines, with 39 of them fixed by the developers. Out of 44 bugs, 22 have security implications. The bugs found by TemuJs span a wide range of components in the JS engines. Our findings demonstrate the significant bug-finding capability of TemuJs for JS engines.

6.3 RQ2: Comparison with Baselines

Comparison Baselines. We compare TemuJs with 5 state-of-the-art JavaScript (JS) engine fuzzers, including Fuzzilli [Groß et al. 2023], DIE [Park et al. 2020], Dumpling [Wachter et al. 2025a], Montage [Lee et al. 2020], and OptFuzz [Wang et al. 2024]. These baselines were selected to provide a comprehensive comparison across different fuzzing paradigms. Each fuzzer employs a unique strategy to explore the JS engine's behaviors and find bugs. Fuzzilli uses static single-assignment (SSA)-based IR and IR-level mutations to generate semantically valid JS programs, focusing on JIT stress-testing. DIE employs structure and type-preserving mutations on regression tests to find new bugs. Dumpling is a differential fuzzer comparing optimized and unoptimized execution to detect misoptimizations. OptFuzz uses a JIT-optimization-driven path scheduling strategy. Montage leverages a language model to generate diverse test cases by manipulating ASTs. By comparing against such a diverse set of fuzzers, we aim to demonstrate the unique strengths and contributions of TemuJs in exploring a broader and more challenging space of JS engine behaviors, and its ability to uncover bugs that may be missed by existing techniques.

Comparison Setup. We use the latest available version of the compared fuzzers to ensure that we use their latest features. Fuzzilli is undergoing regular updates; we use its code base with commit `d4796e`, which is the latest version at the time of writing. Dumpling uses customized instrumentation to instrument the JIT de-optimization infrastructure of V8. Since the instrumentation is specific to V8 and it takes significant efforts to port it to other JS engines, we only evaluate Dumpling on V8.[3]

[3]Two experts in JS engine fuzzing (also paper authors), attempted to port Dumpling to SpiderMonkey and JavaScriptCore. However, Dumpling requires modifying the de-optimization infrastructure, which is fundamentally different across different

Table 2. Comparison of the number of bugs found by baselines, TemuJs and its variants over 45 days.

JS Engine	Fuzzilli	DIE	Dumpling	Montage	OptFuzz	TemuJs	TemuJs^d	TemuJs^c
V8	9	4	6	1	NA	11	10	7
SpiderMonkey	1	0	NA	0	NA	3	1	0
JavaScriptCore	3	4	NA	1	5	6	5	3
Total	13	8	6	2	5	20	16	10

OptFuzz also requires dedicated instrumentation of JS engines. Through the authors' response to our inquiry, we learn that OptFuzz is currently only available for JavaScriptCore. We thus evaluate OptFuzz on JavaScriptCore only. We use their default configurations and seeds for all fuzzers in our assessment. In terms of the initial corpus for fuzzers like Montage and DIE, all the baseline fuzzers include regression test suites from JS engines, i.e., the same set of test cases from which we extract the templates. In other words, the seeds of baseline fuzzers are super sets of seeds of TemuJs. We thus deem the comparison fair and meaningful.

Coverage Comparison. We measure the edge coverage of TemuJs and baselines on the latest version of tested JS engines. We run the fuzzer for 72 hours and repeat the experiments 5 times to mitigate the influence of randomness. For TemuJs and all the baselines, we use 10 dedicated CPU cores for each fuzzer. Each fuzzer employs different instrumentation techniques — DIE uses AFL instrumentation, OptFuzz transforms the program's LLVM IR, and Dumpling modifies the JavaScript engine's source code for customized instrumentation. To ensure consistent measurement for all fuzzers, we collect edge coverage using the standard, unmodified build of each JavaScript engine. Also, we find that Dumpling injects customized native call `%EnableFrameDumping()` into the testing cases. We remove these native calls before measuring the coverage to prevent errors.

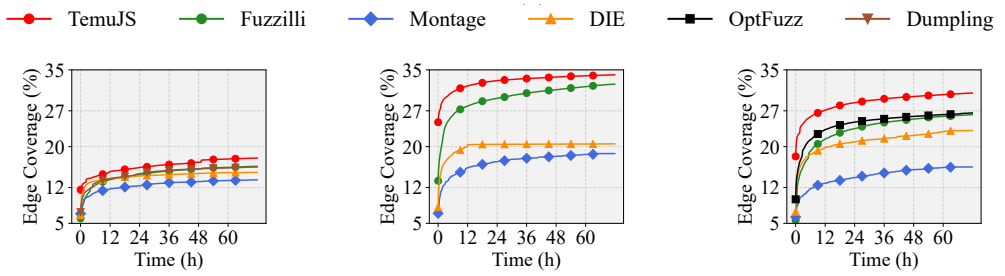
We show the edge coverage trend of TemuJs and baselines in Fig. 6. TemuJs achieves the highest coverage on all the tested JS engines. After 72 hours, the edge coverage of TemuJs on V8, SpiderMonkey, and JavaScriptCore is 17.8%, 34.0%, and 30.5%, respectively. Compared to the second-best fuzzer on each of the three JS engines, TemuJs outperforms by 10.0% on V8, 5.7% on SpiderMonkey, and 2.0% on JavaScriptCore, in terms of the relative coverage increase. On average, the edge coverage gain of TemuJs over all baselines across three JS engines is 10.3%[4]. The results indicate that TemuJs can explore a broader and more meaningful space of program behaviors compared to existing techniques, leading to higher edge coverage.

Considering the large code bases of the JS engines (around a million lines of code), the number of newly covered edges by TemuJs is significant. The results demonstrate the superior performance of the proposed template-based approach in fuzzing JS engines. By mutating high-level abstractions, TemuJs can explore deeper and more diverse paths in JS engines, leading to higher coverage.

Comparison of Bug Findings. We compare the bugs found by TemuJs and the baselines in Table 2. We run TemuJs and baselines for 45 days and report the number of unique bugs found in the JS engines. TemuJs consistently outperforms all baselines across the three JavaScript engines, discovering a total of 20 unique bugs. The second-best fuzzer, Fuzzilli, finds 13 bugs, 7 fewer than TemuJs. The results demonstrate the effectiveness of TemuJs in finding bugs in JS engines. The unique high-level template fuzzing approach of TemuJs enables it to mutate high-level semantics of JS programs, thus leading to the discovery of more bugs compared to existing techniques.

engines. As a result, porting Dumpling to other JS engines is infeasible without significant efforts in re-engineering Dumpling on other engines (the patch of Dumpling to V8 requires thousands of lines of code [Wachter et al. 2025b]).

[4]The coverage gain is computed as the harmonic mean of the relative coverage increase across the three JS engines and all baselines. We use the harmonic mean to reduce the impact of outliers [Komić 2011], which arise due to some fuzzers having significantly lower coverage.



(a) Coverage on V8. (b) Coverage on SpiderMonkey. (c) Coverage on JavaScriptCore.

Fig. 6. Edge coverage trend on the evaluated fuzzers for 72hr. The missing lines indicate that the fuzzer is not available for the corresponding JS engine — by the time of writing, Dumpling only supports V8, and OptFuzz only supports JavaScriptCore.

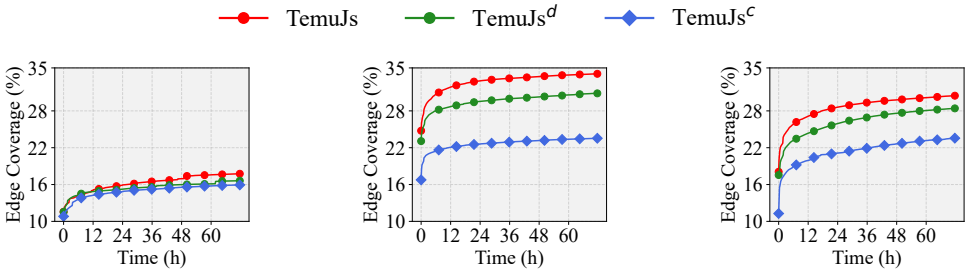
Answer to RQ2: Compared to state-of-the-art JS engine fuzzers, TemuJs achieves a 10.3% relative increase of edge coverage on average. The higher coverage can be attributed to the unique high-level template fuzzing approach of TemuJs, which explores a broader and more meaningful space of program behaviors.

6.4 RQ3: Component Effectiveness

To understand the effectiveness of the individual design components of TemuJs, we conduct an ablation study. We evaluate two key components contributing to TemuJs's performance: ① our data flow-aware template extraction strategy, and ② the design of template mutation. By comparing these components against baseline alternatives, we aim to understand their individual impact on the fuzzer's effectiveness in generating diverse test cases and uncovering bugs.

Ablation Study Setup. To evaluate the effectiveness of the data flow-aware template extraction strategy, we disable the data flow analysis and adopt a pure random strategy to substitute concrete expressions with placeholders in the extracted templates. We refer to this variant as TemuJs^d. To evaluate the influence of template mutation, we replace it with mutations on concrete programs. Specifically, we directly generate testing JS programs from the extracted templates (without template mutation) and then apply the same mutation operators to these concrete programs. We refer to this variant as TemuJs^c. We compare the performance of TemuJs with TemuJs^d and TemuJs^c with the same experimental settings as in Section 6.2.

Coverage Trend for Ablated Components. We run TemuJs, TemuJs^d, and TemuJs^c on the three JS engines for 72 hours with 10 cores, and measure the edge coverage. The coverage trend is illustrated in Fig. 7. TemuJs is shown to be more effective than the other two variants in terms of coverage. The variant TemuJs^d is the second best, with coverage lower than TemuJs by 1% to 3% on the three JS engines. When randomly extracting templates without considering the data flow, TemuJs^d fails to focus on extracting the promising code snippets as templates, and thus wastes resources on less impactful code regions. This leads to a slower exploration of the JS engine code space, hindering its ability to discover new and relevant program behaviors compared to TemuJs, which leverages data flow information for more targeted template extraction. Compared to TemuJs, the edge coverage of TemuJs^c is 2%, 10%, and 7% lower on V8, SpiderMonkey, and JavaScriptCore, respectively. By mutating concrete programs, TemuJs^c is constrained to the specific values and contexts present in those programs, making it harder to trigger novel behaviors or edge cases that



(a) Coverage on V8. (b) Coverage on SpiderMonkey. (c) Coverage on JavaScriptCore.

Fig. 7. Edge coverage trend of TemuJs, TemuJs^d, and TemuJs^c on the evaluated JS engines.

Table 3. Impact of different design choices on fuzzing effectiveness. We report the change in edge coverage (%) and the number of unique bugs found (#) relative to the default TemuJs configuration across the three JavaScript engines during 72 hours. The default setting of TemuJs achieves 17.8% coverage on V8, 34.0% on SpiderMonkey, and 30.5% on JavaScriptCore, with 5, 1, and 2 bugs found, respectively.

Parameter Set	Setting	V8		SpiderMonkey		JavaScriptCore	
		Cov. Δ%	Bugs Δ#	Cov. Δ%	Bugs Δ#	Cov. Δ%	Bugs Δ#
Sampling Strategy	Uniform (Default)	0	0	0	0	0	0
	Gaussian	+0.2	0	+0.2	0	-0.2	0
	Exponential	-0.4	0	-0.1	0	-0.5	0
Heuristics	Def-Use (Default)	0	0	0	0	0	0
	Dataflow Path Length	-0.3	-1	-0.5	-1	-0.2	0
	#Branches	-0.6	-1	-1.5	-1	-1.1	-1
	#Func Calls	-0.5	-2	-1.2	-1	-0.9	-1
	Combined	+0.3	0	+0.7	0	+0.5	+1
	Random	-1.1	-2	-3.1	-1	-2.1	-1
Mutation Operators	Insertion	-2.3	-2	-2.9	-1	-2.6	-1
	Deletion	-3.8	-5	-6.3	-1	-5.6	-2
	Substitution	-2.2	-2	-3.3	-1	-2.9	-1
	Fusion	-3.1	-2	-5.2	-1	-4.8	-1
	Splicing	-2.5	-2	-4.5	-1	-3.9	0

TemuJs, with its template-based approach, can more readily achieve. This reduced diversity in generated test cases ultimately results in lower coverage compared to the full TemuJs.

Bug Finding Comparison for Ablated Components. We compare the number of bugs found by TemuJs, TemuJs^d, and TemuJs^c in the last three columns of Table 2. Over 45 days, TemuJs^d and TemuJs^c find 16 and 10 bugs, respectively, while TemuJs discovers 20 bugs. TemuJs outperforms TemuJs^d by 4 bugs and TemuJs^c by 10 bugs. The results indicate that the data flow-aware template extraction strategy and the template mutation design are both crucial to the effectiveness of TemuJs. Without the data flow analysis or the template mutation, deeper bugs in the JS engines are harder to trigger, leading to fewer bugs found by the fuzzer. The results demonstrate the importance of our high-level template-based fuzzing approach in triggering diverse behaviors in JS engines.

Impact of Design Choices. We additionally vary design choices of TemuJs and measure their impact. We ran experiments on three JS engines in the same setup as in Section 6.3, i.e., 72 hours with 10 cores. The results are summarized in Table 3. We explore the following design choices:

- *Sampling Strategy:* TemuJs by default uses uniform sampling in substituting concrete expressions with placeholders. We additionally explore the effect of using Gaussian sampling and exponential sampling. We set the standard deviation of the Gaussian distribution to 0.1 and the exponential distribution to 0.3. We observe that the Gaussian sampling achieves slightly higher coverage on V8 and SpiderMonkey, while the exponential sampling leads to lower coverage on all three JS engines. Nonetheless, all three sampling strategies yield the same bug finding results. The results indicate that uniform sampling is a robust choice for TemuJs and achieves a good balance between coverage and bug finding.
- *Heuristics for Data Flow-Aware Template Extraction:* We evaluate the impact of different heuristics in Section 4.2. The default heuristic is based on variable def-use. We also explore other data flow metrics such as data flow path length, number of branches, and number of function calls. We find that these metrics alone lead to coverage drop and fewer bug findings compared to the default def-use heuristic. We hypothesize that this could be due to the fact that the Just-in-Time (JIT) compilation in JS engines heavily optimizes based on def-use chains of variables. We also experiment with a combined heuristic that uses all the metrics. It achieves better coverage and bug finding results than the individual metrics. This is expected, as the combined heuristic captures a broader range of data flow characteristics. Users can choose to use the combined heuristic for better performance, or the def-use heuristic for a more lightweight implementation. We note that data flow information is crucial for template extraction to focus on the most promising code regions — without any heuristics, i.e., randomly selecting code fragments, the coverage and bug finding results drop significantly.
- *Mutation Operators:* We apply individual operators of Section 4.3 in isolation and evaluate their effectiveness. We observe that enabling all mutators, i.e., the default mode of TemuJs, yields the best results compared to isolated mutators. The insertion and substitution operators are the most effective ones, as they lead to lower coverage drop compared to the default TemuJs configuration. These two operators introduce new code fragments into the templates, thus leading to unexpected, novel behaviors that do not exist in the original seed templates.
- *Context Element Selection in Concretization:* We evaluate the impact of different context element selection strategies in Algorithm 5. The default strategy is to randomly select an element from the context that matches the placeholder's type. We also explored a strategy guided by data flow complexity. However, this alternative yields statistically similar results to the default random selection. We hypothesize that the structural diversity introduced during template mutation is the primary driver of TemuJs's effectiveness. While the concretization process is essential for generating valid test cases, the high-level template structure already contains the search space. Consequently, optimizing context element selection in concretization offers marginal returns, as it does not significantly alter the fundamental program logic established by the template.

Answer to RQ3: The template extraction strategy and the template mutation design are both essential to the effectiveness of TemuJs. Without the two designs, the edge coverage reduces by 1% to 10% and the total number of bugs found decreases by up to 10.

7 Discussion

Extension to Other JS Engines. TemuJs is evaluated on three widely-used JS engines (V8, SpiderMonkey, and JavaScriptCore). A potential concern with the generalizability of TemuJs is its extensibility to other JS engines. To address such a concern, TemuJs is designed to be agnostic to a specific implementation of the tested JS engine. The core of TemuJs relies on template extraction and mutation, which operate on an abstract representation of JavaScript code. The JS language

conforms to the ECMAScript standard [Ecma 2025], ensuring a degree of uniformity across different engines. While each engine may have its own unique optimizations and internal structures, they all must adhere to the ECMAScript specification. This adherence allows TemuJs to generate valid JavaScript code that can be executed across different engines. We leave it as future work to run large-scale fuzzing campaigns with TemuJs on a broader range of JS engines, including those used in embedded systems and IoT devices, to further assess its generalizability and effectiveness.

Fuzzing vs. Verification. While formal verification offers the promise of mathematically proving the correctness of software, its application to complex systems like JS engines faces significant challenges, making testing a more practical approach for ensuring reliability. Verification requires creating precise formal models of both the system and its desired properties, a task that becomes exceedingly difficult given the intricate semantics of JS and the sophisticated optimizations performed by JS engines. Although fuzzing cannot guarantee the absence of bugs, it offers a more scalable and cost-effective approach. Additionally, developers can leverage our discovered error-triggering inputs and their stack traces to debug and patch the vulnerabilities. In line with other works [Bernhard et al. 2022; Le et al. 2014, 2015; Yang et al. 2011] on reliability enhancement for complex software, TemuJs adopts fuzzing as its main technique.

Generation of Complex JS Language Features. The abstraction and concretization mechanism in TemuJs enables the generation and preservation of complex JS language features that are difficult to synthesize directly. Abstraction in TemuJs preserves hard-to-model JS language features by leaving such constructs (e.g., `Reflect` in Fig. 2f, `await` in asynchronous functions) untouched during template extraction, rather than attempting to model them as templates. This ensures that advanced or dynamic features remain present in the generated test cases. The template mutation component then mutates these preserved constructs, enabling novel data/control flow combinations and interactions. Finally, concretization instantiates the templates, producing executable JS programs that retain and exercise these complex language features, thereby increasing the likelihood of uncovering subtle engine bugs. We note that users who want to generate test cases that target specific JS language features can provide seed programs that contain such features, which will be preserved in the templates and concretized test cases.

8 Related Work

Compiler Testing. Compiler testing aims to uncover bugs in compilers to enhance their reliability. Compiler testing faces the oracle problem, i.e., lacking ground truth for deciding the expected outputs for compiling input programs. Metamorphic testing (MT) [Chen et al. 2020; Li et al. 2024a; Liu and Wang 2020; Wong et al. 2024, 2022] addresses this problem by asserting invariant properties for tested compilers. MT has been applied to a wide range of compilers, including JVM [Li et al. 2023], C/C++ compilers [Le et al. 2014, 2015; Sun et al. 2016], privacy-enhancing technology compilers [Li et al. 2024b; Xiao et al. 2025a], deep learning compilers [Ma et al. 2023; Xiao et al. 2022], graphics shader compilers [Donaldson et al. 2017; Xiao et al. 2023], and WebAssembly compilers [Liu et al. 2023]. Differential testing (DT) [McKeeman 1998] compares the outputs from different compilers to identify bugs, and has found successful applications in GCC/LLVM [Theodoridis et al. 2022; Yang et al. 2011], JVM [Chen et al. 2016], and graphics shader compilers [Xiao et al. 2025b]. Similar to other compiler testing works, TemuJs also crafts testing programs as the testing inputs. However, since JS engines contain both interpreters and compilers, we propose a template extraction and mutation for effective fuzzing of JS engines.

Java Virtual Machine Fuzzing. Java Virtual Machine (JVM) fuzzing is critical for uncovering bugs in JVM implementations, which are essential for platform-independent Java program execution. Existing tools employ diverse strategies for this: Classfuzz [Chen et al. 2016] leverages Markov Chain Monte Carlo (MCMC) sampling and coverage as guidance, Classming [Chen et al. 2019] crafts

mutation rules to generate valid Java bytecode, and JavaTailor [Zhao et al. 2022]/Jetris [Zhao et al. 2024] incorporate historical bug-triggering cases to generate new testing inputs. JITFuzz [Wu et al. 2023] designs optimization-activating mutators to trigger JIT optimization bugs. JOptFuzzer [Jia et al. 2023] jointly mutates both Java programs and compilation options. Artemis [Li et al. 2023] triggers fine-grained JIT optimizations and detects inconsistencies between different optimization levels. It would be promising to extend TemuJs to test JVM implementations, provided with the necessary engineering efforts to adapt to the JVM-specific features. While JAttack [Zang et al. 2023] and LeJit [Zang et al. 2024] also employ template-based fuzzing to detect bugs in JVMs, their templates are limited to simple syntax-level patterns, whereas TemuJs defines templates at the semantic level to capture the high-level behaviors of JS programs. In addition, template generation for JVM relies on type information, which is not applicable to dynamic languages like JavaScript. Also, TemuJs mutates the templates to enlarge the mutation space, while JAttack and LeJit do not perform mutations and thus may generate less diverse test cases.

JavaScript Engine Fuzzing. Existing JS engine fuzzing works generate test cases at different granularities. TokenFuzz [Salls et al. 2021] and CovRL-Fuzz [Eom et al. 2024] mutate tokens, whereas Jsfunfuzz [Mozilla 2023], LangFuzz [Holler et al. 2012], CodeAlchemist [Han et al. 2019], Montage [Lee et al. 2020], and Superion [Wang et al. 2019] mutate ASTs. DIE [Park et al. 2020] and SoFi [He et al. 2021] mutate ASTs annotated with type information. However, all of these approaches mutate on the syntax level and result in low semantic diversity of the generated JS code. Some other works design dedicated IRs to capture the semantics of JS programs for more effective fuzzing. FuzzJIT [Wang et al. 2023] and Fuzzilli [Groß et al. 2023] generate/mutate JS programs in SSA forms, and FuzzFlow [Xu et al. 2024] designs graph IRs to represent control/data flow of JS programs. While these works capture the semantics of JS engines in a finer-grained manner, they are still limited by the mutation space to conform to the low-level details of JS programs. TemuJs, instead, extracts templates to abstract the semantics of seed programs and mutates their high-level abstractions. OptFuzz [Wang et al. 2024] improves fuzzing efficiency by designing better coverage feedback. Some works focus on detecting logic bugs in the JIT compilation components of JS engines. Comfort [Ye et al. 2021] and Jest [Park et al. 2021] apply conformance testing to detect inconsistencies between JS engine execution results and the ECMAScript standard. TurboTV [Kwon et al. 2024] employs fuzzing as a practical approach to validate JIT compilers. JITpicker [Bernhard et al. 2022] applies differential testing [McKeeman 1998] techniques to uncover inconsistencies between JIT-compiled and interpreted code. FuzzJIT [Wang et al. 2023] employs JIT-aware comparison rules and checks for deviation in behaviors. Dumpling [Wachter et al. 2025a] adopts fine-grained deviation detection through customized instrumentation of JS engines. Instead of detecting logic bugs, our work aims to detect vulnerabilities in JS engines, and therefore has an orthogonal scope with these works.

9 Conclusion

We propose TemuJs, a mutation-based testing tool designed specifically to uncover bugs in JavaScript engines. By coupling template extraction and mutation techniques to abstract the high-level semantics of JavaScript programs, TemuJs effectively explores the complex behaviors of JavaScript engines and finds real-world bugs. Our evaluation on V8, SpiderMonkey, and JavaScriptCore found 44 bugs, including 22 security vulnerabilities, demonstrating TemuJs's high effectiveness and real-world impact on JavaScript engine reliability. It also serves as valuable inspiration for future research on testing and securing JavaScript engines.

Data-Availability Statement

This project is under a commercial contract for third-party fuzzing, which limits the release of the full source code and error-triggering inputs. We will open-source the tool and release the

details of the error-triggering JS programs as soon as we obtain necessary permissions. Nonetheless, after preliminary negotiation with the company, we are able to share a prototype of TemuJs at https://sites.google.com/view/temujs.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. The HKUST authors are supported in part by a RGC GRF grant under the contract 16214723. We also thank Prof. Zhendong Su at ETH Zürich for his valuable discussions with Dongwei Xiao regarding the ideas presented in this paper, which took place during Dongwei Xiao's visit. We additionally thank Samuel Groß and Carl Smith at Google for their valuable feedback during the initial stages of this work.

References

Apple. 2025a. Dispatch | Apple Developer Documentation. https://developer.apple.com/documentation/DISPATCH.

Apple. 2025b. JavaScriptCore. https://developer.apple.com/documentation/javascriptcore.

Apple. 2025c. JavaScriptCore regression tests. https://github.com/WebKit/WebKit/tree/main/JSTests/stress.

Apple. 2025d. Swift.org - Welcome to Swift.org. https://www.swift.org/.

Lukas Bernhard, Tobias Scharnowski, Moritz Schloegel, Tim Blazytko, and Thorsten Holz. 2022. JIT-picking: Differential fuzzing of JavaScript engines. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 351–364. doi:10.1145/3548606.3560624

Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 2020. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543* (2020).

Yuting Chen, Ting Su, and Zhendong Su. 2019. Deep differential testing of JVM implementations. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1257–1268. doi:10.1109/ICSE.2019.00127

Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 85–99. doi:10.1145/2908080.2908095

Gemma Crawford. 2024. A Critical Vulnerability in Chrome's V8 JavaScript Engine - CommuniCloud. https://www.communicloud.com/blog/chrome-v8-security-exploit/.

Cyble. 2024. Urgent Fix Needed For Chrome V8 Engine CVE-2024-7965. https://cyble.com/blog/high-risk-cve-2024-7965-vulnerability-in-chromes-v8-engine-requires-quick-fix/.

Dev Learning Daily. 2024. Role of JavaScript engines in browser architecture. https://learningdaily.dev/role-of-javascript-engines-in-browser-architecture-ba63c3dceb05.

Alastair F Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. 2017. Automated testing of graphics shader compilers. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–29. doi:10.1145/3133917

Ecma. 2025. ECMAScript 2025 Language Specification. https://tc39.es/ecma262/.

Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. 2024. Fuzzing JavaScript Interpreters with Coverage-Guided Reinforcement Learning for LLM-Based Mutation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1656–1668. doi:10.1145/3650212.3680389

Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. doi:10.5555/3488877.3488887

Google. 2016. V8 parser fuzzer. https://source.chromium.org/chromium/chromium/src/+/main:v8/test/fuzzer/parser.cc.

Google. 2025a. A Brief JavaScriptCore RCE Story. https://qriousec.github.io/post/jsc-uninit/.

Google. 2025b. Code base of V8. https://chromium.googlesource.com/v8/v8.git/.

Google. 2025c. Public v8CTF sbumissions. https://docs.google.com/spreadsheets/d/e/2PACX-1vTWvO0tFNl8fJbOmTV1nwGJi4fAy5pDg-6DsHARRubj8I6c7_11RQ36Jv735zj9EQggz6AWjAOaebJh/pubhtml.

Google. 2025d. V8 JavaScript engine. https://v8.dev/.

Google. 2025e. V8 regression tests. https://github.com/v8/v8/tree/main/test/mjsunit/regress.

GraffersID. 2025. What is JavaScript And Why JavaScript Is So Popular? https://graffersid.com/why-javascript-is-popular/.

Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. 2023. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In *Proceedings 2023 Network and Distributed System Security Symposium*. Internet Society, San Diego, CA, USA. doi:10.14722/ndss.2023.24290

HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. 2019. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines.. In *NDSS*. doi:10.14722/ndss.2019.23263

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 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2229–2242. doi:10.1145/3460120.3484823

Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L Hosking. 2021. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*. 230–243. doi:10.1145/3460319.3464795

Ariya Hidayat. 2025. Esprima. https://esprima.org/.

Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*. 445–458. doi:10.5555/2362793.2362831

Haoxiang Jia, Ming Wen, Zifan Xie, Xiaochen Guo, Rongxin Wu, Maolin Sun, Kang Chen, and Hai Jin. 2023. Detecting JVM JIT compiler bugs via exploring two-dimensional input spaces. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 43–55. doi:10.1109/ICSE48619.2023.00016

Jasmin Komić. 2011. *Harmonic Mean*. Springer Berlin Heidelberg, Berlin, Heidelberg, 622–624. doi:10.1007/978-3-642-04898-2_645

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 IEEE/ACM 46th International Conference on Software Engineering*. 1–12. doi:10.1145/3597503.3639189

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86. doi:10.1109/CGO.2004.1281665

Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices* 49, 6 (2014), 216–226. doi:10.1145/2666356.2594334

Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. *ACM SIGPLAN Notices* 50, 10 (2015), 386–399. doi:10.1145/2858965.2814319

Suyoung Lee, HyungSeok Han, Sang Kil Cha, and Sooel Son. 2020. Montage: A Neural Network Language Model-Guided JavaScript Engine Fuzzer. In *29th USENIX Security Symposium (USENIX Security 20)*. 2613–2630. doi:10.5555/3489212.3489359

Cong Li, Yanyan Jiang, Chang Xu, and Zhendong Su. 2023. Validating JIT compilers via compilation space exploration. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 66–79. doi:10.1145/3715102

Yichen Li, Dongwei Xiao, Zhibo Liu, Qi Pang, and Shuai Wang. 2024b. Metamorphic testing of secure multi-party computation (mpc) compilers. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1216–1237. doi:10.1145/3643781

Zongjie Li, Zhibo Liu, Wai Kin Wong, Pingchuan Ma, and Shuai Wang. 2024a. Evaluating C/C++ vulnerability detectability of query-based static application security testing tools. *IEEE Transactions on Dependable and Secure Computing* 21, 5 (2024), 4600–4618. doi:10.1109/TDSC.2024.3354789

Zhibo Liu and Shuai Wang. 2020. How far we have come: Testing decompilation correctness of C decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 475–487. doi:10.1145/3395363.3397370

Zhibo Liu, Dongwei Xiao, Zongjie Li, Shuai Wang, and Wei Meng. 2023. Exploring missed optimizations in webassembly optimizers. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 436–448. doi:10.1145/3597926.3598068

Haoyang Ma, Qingchao Shen, Yongqiang Tian, Junjie Chen, and Shing-Chi Cheung. 2023. Fuzzing deep learning compilers with hirgen. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 248–260. doi:10.1145/3597926.3598053

William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

Mozilla. 2023. GitHub - MozillaSecurity/funfuzz: A collection of fuzzers in a harness for testing the SpiderMonkey JavaScript engine. https://github.com/MozillaSecurity/funfuzz/tree/master.

Mozilla. 2025a. Nullish coalescing operator (??) - JavaScript | MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Nullish_coalescing.

Mozilla. 2025b. Reflect - JavaScript | MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Reflect.

Mozilla. 2025c. SpiderMonkey. https://spidermonkey.dev/.

Mozilla. 2025d. SpiderMonkey regression tests. https://github.com/mozilla/gecko-dev/tree/master/js/src/jit-test/tests.

Jihyeok Park, Seungmin An, Dongjun Youn, Gyeongwon Kim, and Sukyoung Ryu. 2021. Jest: N+ 1-version differential testing of both javascript engines and specification. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 13–24. doi:10.1109/ICSE43902.2021.00015

Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1629–1642. doi:10.1109/SP40000.2020.00067

Pramod Pawar and Rohan Jambhale. 2024. JavaScript Statistics By Usage and Facts. https://electroiq.com/stats/javascript-statistics/.

Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. 2021. Token-Level Fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2795–2809. https://www.usenix.org/conference/usenixsecurity21/presentation/salls

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*. 309–318. doi:10.5555/2342821.2342849

StatCounter. 2024. Browser Market Share Worldwide | Statcounter Global Stats. https://gs.statcounter.com/browser-market-share.

Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 849–863. doi:10.1145/2983990.2984038

Coding Temple. 2024. Why is JavaScript So Popular? Key Factors Explained | Coding Temple. https://www.codingtemple.com/blog/why-has-javascript-become-so-popular-exploring-key-factors/.

Theodoros Theodoridis, Manuel Rigger, and Zhendong Su. 2022. Finding missed optimizations through the lens of dead code elimination. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 697–709. doi:10.1145/3503222.3507764

TrueFort. 2024. 8 Chrome Vulnerabilities that Caused Risk in 2024. https://truefort.com/8-dangerous-chrome-vulnerabilities/.

Liam Wachter, Julian Gremminger, Christian Wressnegger, Mathias Payer, and Flavio Toffalini. 2025a. DUMPLING: Fine-grained Differential JavaScript Engine Fuzzing. In *Proceedings 2025 Network and Distributed System Security Symposium*. Internet Society. doi:10.14722/ndss.2025.241411

Liam Wachter, Julian Gremminger, Christian Wressnegger, Mathias Payer, and Flavio Toffalini. 2025b. Dumpling V8 patch. https://github.com/two-heart/dumpling-artifact-evaluation/blob/main/bug_finding_and_overhead/dumpling_v8.patch.

Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 724–735. doi:10.1109/ICSE.2019.00081

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 *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 865–882. doi:10.5555/3698900.3698949

Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. 2023. FuzzJIT: Oracle-Enhanced Fuzzing for JavaScript Engine JIT Compiler. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1865–1882. doi:10.5555/3620237.3620342

Wai Kin Wong, Huaijin Wang, Zongjie Li, and Shuai Wang. 2024. Binaug: Enhancing binary similarity analysis with low-cost input repairing. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3623328

Wai Kin Wong, Huaijin Wang, Pingchuan Ma, Shuai Wang, Mingyue Jiang, Tsong Yueh Chen, Qiyi Tang, Sen Nie, and Shi Wu. 2022. Deceiving deep neural networks-based binary code matching with adversarial programs. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 117–128. doi:10.1109/ICSME55016.2022.00019

Wai Kin Wong, Daoyuan Wu, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2025a. DecLLM: LLM-Augmented Recompilable Decompilation for Enabling Programmatic Use of Decompiled Code. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 1841–1864. doi:10.1145/3728958

Wai Kin Wong, Dongwei Xiao, Cheuk Tung Lai, Yiteng Peng, Daoyuan Wu, and Shuai Wang. 2025b. List of bugs found by TemuJS. https://sites.google.com/view/temujs/bugs.

Mingyuan Wu, Minghai Lu, Heming Cui, Junjie Chen, Yuqun Zhang, and Lingming Zhang. 2023. Jitfuzz: Coverage-guided fuzzing for jvm just-in-time compilers. In *2023 IEEE/acm 45th international conference on software engineering (icse)*. IEEE, 56–68. doi:10.1109/ICSE48619.2023.00017

Dongwei Xiao, Zhibo Liu, Yiteng Peng, and Shuai Wang. 2025a. Mtzk: Testing and exploring bugs in zero-knowledge (zk) compilers. In *NDSS*. doi:10.14722/ndss.2025.230530

Dongwei Xiao, Zhibo Liu, and Shuai Wang. 2023. Metamorphic shader fusion for testing graphics shader compilers. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2400–2412. doi:10.1109/ICSE48619.2023.00201

Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Metamorphic Testing of Deep Learning Compilers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 1 (2022), 1–28. doi:10.1145/3508035

Dongwei Xiao, Shuai Wang, Zhibo Liu, Yiteng Peng, Daoyuan Wu, and Zhendong Su. 2025b. Divergence-Aware Testing of Graphics Shader Compiler Back-Ends. *Proceedings of the ACM on Programming Languages* 9, PLDI (2025), 1367–1391. doi:10.1145/3729305

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 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. Association for Computing Machinery, New York, NY, USA, 3734–3748. doi:10.1145/3658644.3690336

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294. doi:10.1145/1993316.1993532

Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. 2021. Automated conformance testing for JavaScript engines via deep compiler fuzzing. In *Proceedings of the 42nd ACM SIGPLAN international conference on programming language design and implementation*. 435–450. doi:10.1145/3453483.3454054

Michal Zalewski. 2025. afl-cmin. https://github.com/AFLplusplus/AFLplusplus/blob/stable/afl-cmin.

Zhiqiang Zang, Fu-Yao Yu, Aditya Thimmaiah, August Shi, and Milos Gligoric. 2024. Java JIT testing with template extraction. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1129–1151. doi:10.1145/3643777

Zhiqiang Zang, Fu-Yao Yu, Nathan Wiatrek, Milos Gligoric, and August Shi. 2023. JAttack: Java JIT testing using template programs. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 6–10. doi:10.1109/ICSE-Companion58688.2023.00014

Yingquan Zhao, Zan Wang, Junjie Chen, Ruifeng Fu, Yanzhou Lu, Tianchang Gao, and Haojie Ye. 2024. Program Ingredients Abstraction and Instantiation for Synthesis-based JVM Testing. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 3943–3957. doi:10.1145/3658644.3690366

Yingquan Zhao, Zan Wang, Junjie Chen, Mengdi Liu, Mingyuan Wu, Yuqun Zhang, and Lingming Zhang. 2022. History-driven test program synthesis for JVM testing. In *Proceedings of the 44th International Conference on Software Engineering*. 1133–1144. doi:10.1145/3510003.3510059

Received 2025-03-25; accepted 2025-08-12