OBsmith: LLM-powered JavaScript Obfuscator Testing

SHAN JIANG, The University of Texas at Austin, USA
CHENGUANG ZHU, The University of Texas at Austin, USA
SARFRAZ KHURSHID, The University of Texas at Austin, USA

JavaScript obfuscators are widely deployed to protect intellectual property and resist reverse engineering, yet their correctness has been largely overlooked compared to performance and resilience. Existing evaluations typically measure resistance to deobfuscation, leaving the critical question of whether obfuscators preserve program semantics unanswered. Incorrect transformations can silently alter functionality, compromise reliability, and erode security—undermining the very purpose of obfuscation. To address this gap, we present OBsmith, a novel framework to systematically test JavaScript obfuscators using large language models (LLMs). OBsmith leverages LLMs to generate program sketches—abstract templates capturing diverse language constructs, idioms, and corner cases—which are instantiated into executable programs and subjected to obfuscation under different configurations. Besides LLM-powered sketching, OBsmith also employs a second source: automatic extraction of skeletons from real programs. This extraction path enables more focused testing of project-specific features and lets developers inject domain knowledge into the resulting test cases. OBsmith uses two techniques to derive test oracles: *(i)* differential testing, which treats the original program as ground truth and instruments control- and data-flow to detect semantic divergences, failures, or crashes; and *(ii)* metamorphic testing, which applies semantics-preserving transformations to the original program and checks if obfuscation violates expected behavior.

We evaluate OBsmith on two widely used obfuscators, Obfuscator.IO and JS-Confuser, generating 600 sketches using six popular LLMs. OBsmith fills these sketches and generates over 3,000 candidate programs and obfuscates them across seven obfuscation configurations. OBsmith uncovers 11 previously unknown correctness bugs. Under an equal program budget, five general purpose state-of-the-art JavaScript fuzzers (FuzzJIT, Jsfunfuzz, Superion, DIE, Fuzzilli) failed to detect these issues, highlighting OBsmith's complementary focus on obfuscation-induced misbehavior. An ablation shows that all components except our generic MRs contribute to at least one bug class; the negative MR result suggests the need for obfuscator-specific metamorphic relations. Our results also seed discussion on how to balance obfuscation presets and performance cost. We envision OBsmith as an important step towards automated testing and quality assurance of obfuscators and other semantic-preserving toolchains.

CCS Concepts: • **Software and its engineering → Formal software verification**; **Correctness**; **Completeness**; **Software verification**; • **Computing methodologies → Artificial intelligence**; • **General and reference → Metrics**; **Evaluation**.

Additional Key Words and Phrases: Software Testing, JavaScript Obfuscator, Large Language Models, Program Sketching

ACM Reference Format:
Shan Jiang, Chenguang Zhu, and Sarfraz Khurshid. 2024. OBsmith: LLM-powered JavaScript Obfuscator Testing. *J. ACM* 37, 4, Article 111 (August 2024), 28 pages. https://doi.org/XXXXXXX.XXXXXXX

Authors' Contact Information: Shan Jiang, The University of Texas at Austin, Austin, USA, shanjiang@utexas.edu; Chenguang Zhu, The University of Texas at Austin, Austin, USA, cgzhu@utexas.edu; Sarfraz Khurshid, The University of Texas at Austin, Austin, Texas, USA, khurshid@ece.utexas.edu.

ACM 1557-735X/2024/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Software obfuscation is an important technique that makes code difficult to read, analyze, or reverse engineer. This objective is typically achieved through code transformations such as changing code structures, renaming variables to non-descriptive identifiers, introducing misleading or redundant code, restructuring control flows, and encrypting specific data and strings [8, 12, 89]. While obfuscation can serve legitimate purposes, including the protection of proprietary code or sensitive information such as IP addresses [16, 44], it is also widely adopted by malicious actors to conceal the intent of malicious scripts, particularly within web and Android applications [11, 46, 50, 61]. JavaScript, the predominant language for client-side web applications, is particularly susceptible to obfuscation due to its open and accessible nature [10, 13, 20, 49]. Any JavaScript code executed within a browser can be easily viewed and analyzed, rendering it an attractive target for both benign and malicious obfuscation. JavaScript is a versatile language, functioning across both client-side and server-side environments, each characterized by distinct execution contexts, capabilities, and security requirements.

By design, an effective obfuscator is expected to complicate the original source code without altering its intended functionality. The primary goal is to make the code substantially more difficult for reverse engineers and automated analysis tools to understand, thus safeguarding proprietary or sensitive information. Ensuring functional equivalence between obfuscated and original code is critical, as obfuscation may introduce unintended behaviors, potentially undermining software reliability or security. There is still a lack of research for evaluating Obfuscators on pure JavaScript code. Previous research on JavaScript obfuscation largely focuses on evaluating the effectiveness of obfuscation techniques or detecting if code is obfuscated. However, comprehensive testing frameworks that validate both functional correctness and runtime performance impacts of JavaScript obfuscators remain underexplored. Recent research by Skolka et al.[65] offers a comprehensive evaluation of JavaScript minifiers and obfuscators, with particular emphasis on their robustness when applied to client-side scripts. The authors primarily relied on unit test to assess the effectiveness of various obfuscators. Their findings reveal that these tools often exhibit insufficient resilience. A core challenge of client-side script is that client-side JavaScript is inherently dependent on complex factors such as user interactions, asynchronous events, and integration with real-world browser APIs (e.g., DOM, cookies, localStorage). These dependencies are difficult to replicate or comprehensively test using automated methods. As a result, automated source code transformations—such as minification and obfuscation—may inadvertently introduce bugs in client side. Moreover, prior studies have omitted the evaluation of some of the most widely adopted obfuscators, such as Obfuscator.IO[4] and JS-Confuser[3], as indicated by a recent Google survey[30]. This omission constitutes another important research gap, particularly given the pervasive use of these tools in the software industry. To address this limitation, the present study aims to instrument and evaluate pure JavaScript code that is executable in both Node.js and browser environments. Addressing this research gap is crucial to ensuring that JavaScript obfuscators can reliably achieve their intended security and functionality objectives without inadvertently compromising software behavior.

General purpose compiler testing approaches lack of a systematically-enhanced test oracle to capture the JavaScript obfuscator bugs. Off-the-shelf compiler fuzzers assume closed-world, deterministic pipelines whose goal is transparent semantics preservation; they rely on crash oracles or n-version agreement (e.g., GCC vs. Clang). JavaScript obfuscators instead *preserve while disguising* semantics, inject seed-dependent randomness and self-defense checks, and interact with host features that general fuzzers neither generate nor control. Bugs often surface in JS-specific corners—eval function, dynamic code loading, scoping/hoisting, prototype mutations, getters/setters, proxies—combined with obfuscation patterns like control-flow flattening or lazy

string decoding. These properties defeat differential comparisons across tools and make crashes a poor proxy for silent miscompilations. We take the *original program as the ground-truth oracle* and check semantic equivalence—including return values, exceptions, scheduling-sensitive effects, and observable host-API interactions. OBsmith generates JS-centric, event-aware sketches, controls nondeterminism via seed management, executes in sandboxed Node and browser-like environments, and is configuration-aware to sweep obfuscator options. Lightweight instrumentation records variable states and control-flow to localize divergences. This semantics-, host-, and transformation-aware design exposes obfuscation-specific bugs that general compiler fuzzers systematically miss. Obfuscators differ from compilers because they preserve but disguise semantics. They often use tricks (control-flow flattening, dead code) that do not appear in compiler pipelines. Hence, domain-specific sketching/testing is required.

We propose OBsmith, a novel testing framework specifically designed for evaluating JavaScript obfuscators. OBsmith leverages sketching techniques to systematically generate diverse test programs and employs differential testing to identify discrepancies introduced by obfuscation processes. Sketching and differential testing methodologies have previously demonstrated effectiveness in compiler testing, enabling the detection of subtle semantic bugs and inconsistencies. Extensive literature on compiler testing underscores the value and efficacy of integrating domain-specific knowledge into test-case generation. Inspired by these insights, we adopt sketching to incorporate JavaScript-specific domain knowledge into our framework, thereby enhancing the capability of OBsmith to detect functional deviations and performance degradations caused by obfuscators.

OBsmith uses LLM to generate sketches and uses Babel[1] to implement the verification and differential testing mechanism. Trained on extensive textual corpora, LLMs inherently encapsulate rich domain-specific knowledge, making them particularly suitable for generating sketches in automated test-case generation. Leveraging their ability to understand syntactic structures and semantic constraints, LLMs can systematically produce representative code sketches that reflect realistic programming patterns and corner cases. Integrating LLM-based sketch generation into differential testing frameworks enhances test diversity and coverage with low resource, thus enabling the effective discovery of subtle functional inconsistencies and edge-case defects in software, such as those potentially introduced by JavaScript obfuscators.

A critical component of OBsmith is its integrated program generator, which is engineered to rigorously explore different execution path and expose the semantic equivalence between the original source program and its obfuscated counterpart. This step is indispensable, as the intricate nature of obfuscation transformations carries an inherent risk of introducing functional regressions or altering program behavior in subtle ways. To mitigate this risk, OBsmith contains a sketch filling algorithm to generate a comprehensive set of inputs to test different execution paths. OBsmith uses a program enhancer to record global and local variables and record the program's control flow. For each input, OBsmith asserts that the original and obfuscated program versions produce identical outputs and exhibit equivalent observable side-effects. This process provides a strong, empirical guarantee that the core logic and functionality of the application remain unaltered post-obfuscation, thereby enhancing the trustworthiness and practical deployability of our approach.

Unlike the standard differential testing setup used in compiler validation – where multiple variants of a system are run on the same input and their outputs compared for inconsistencies – OBsmith relies on a concrete ground truth. In particular, we treat the original, unobfuscated program as the definitive oracle for correct behavior. This direct comparison against a known-correct baseline allows for a precise and conclusive assessment of the correctness of the obfuscation transformations, ensuring that they have not introduced functional changes.

We evaluate OBsmith on two widely used obfuscators, Obfuscator.IO and JS-Confuser, generating 600 sketches using six popular LLMs. OBsmith fills theese sketches and generates over 3,000

candidate programs and obfuscate them across seven obfuscation configurations. OBsmith uncovers 11 previously unknown correctness bugs. Under an equal program budget, five general purpose state-of-the-art JavaScript fuzzers (FuzzJIT, Jsfunfuzz, Superion, DIE, Fuzzilli) failed to rediscover these issues, highlighting OBsmith's complementary focus on obfuscation-induced misbehavior. We also discuss obfuscation affect on file size, run time, and memory usage in discussion (§5).

To summarize, this paper makes the following contributions:

- **Novelty.** This paper introduces OBsmith, the first LLM-powered framework for systematically testing JavaScript obfuscators. OBsmith introduces an LLM-driven, sketch-based generator (mixing LLM-created and automatically extracted sketches) plus a PL-aware oracle that combines differential testing of outputs/exceptions/termination with lightweight instrumentation and exploratory metamorphic tests.
- **Implementation.** We implement OBsmith with 2 sketch source: (i) leverages multi-agent LLM system to automatically generate diverse sketches and (ii) use existing JavaScript programs automatically extract sketch. To solve oracle problem, OBsmith applies (i) differential testing and metamorphic testing to evaluate the correctness of obfuscation.
- **Real-world Bugs.** We conduct a comprehensive study of two widely used obfuscators across seven configurations and uncover 11 correctness bugs, including silent miscompilations where the obfuscated code changes behavior. All bugs were confirmed by reproducing them in both online and repository versions. These findings highlight the unreliability of obfuscators and provide actionable insights for both developers (to fix issues) and practitioners (to make informed tool choices).

2 Sketch Example

```
let x = NumberLiteral;
let y = NumberLiteral;
let text = "x*y";
console.log(NumberReference)
console.log(text)
let result = eval(text);
console.log(result +
NumberReference);
console.log(text +
NumberReference);
```

```
let x = 10;
let y = 20;
let text = "x*y";
console.log(x)
console.log(text)
let result = eval(text);
console.log(result + x);
console.log(text + y);
```

```
10
x * y
undefined: 1
x * y
^

ReferenceError:
x is not defined
...
```

Fig. 1. A simplified sketch with holes (left), the corresponding concrete program that is input to obfuscators (middle), and the output of obfuscated program created by JS-Confuser which shows its faulty behavior (right).

In this paper, a sketch refers to a "program with holes," where key components - such as expressions, variables, or literals - are left as placeholders to be filled in later. Sketches, also known as "skeletal programs" or "templates," have been shown in previous work to be an effective mechanism for introducing domain knowledge into compiler testing[75, 86, 88]. By abstracting certain program elements, sketches allow for a flexible yet systematic exploration of diverse code scenarios that are effective in exposing compiler bugs. The left part of Figure 1 is an example of sketch with holes (e.g. NumberLiteral, NumberReference) and the middle is the corresponding filled code.

Building upon the sketch definition, OBsmith introduces LLMs to the sketch generation pipeline, uses the power of LLMs to automatically generate program sketches. LLMs, trained on large code corpora, inherently encode a broad range of programming concepts and domain-specific patterns.

This capability allows them to produce high-quality sketches that capture the syntax and structure of real-world programs. By leveraging LLMs for sketch generation, OBsmith automates a previously manual and expertise-driven task, making it possible to efficiently create diverse and domain-informed templates. Afterward, OBsmith uses a program generator to solve the LLM-generated sketches with concrete values or variables, producing executable JavaScript programs that remain faithful to the intended domain constraints.

OBsmith combines LLM-based sketch generation with a program generator, it enables precise and effective testing while reducing manual effort and minimizing the risk of introducing unintended errors during test case generation. By elevating sketches to the core of its workflow and integrating LLMs as a key component, OBsmith sets a new direction for leveraging LLM's domain knowledge in automated testing.

JavaScript obfuscators take two inputs: the program under obfuscation and the obfuscation configuration. Once we obfuscate the concrete program with different obfuscators and different configurations, we got the multiple obfuscated programs. Then we conduct a differential testing on these programs to compare if they have the same functionality. Unlike traditional diff testing in compiler testing, we have ground truth (unobfuscated program). Here the concrete program serve as ground truth and we compare the running result to find bugs. Furthermore, OBsmith enhances programs by recording the program's workflow and data flow, which effectively reflects the program functionality and enables us to easily implement differential testing. An example of found bug is shown in the right part of Figure 1. Here the variable x is reported as undefined during execution (ReferenceError: x is not defined). This error highlights a critical functionality bug introduced by JS-Confuser, demonstrating how sketch-based differential testing effectively detects discrepancies between original program and obfuscated program.

3 OBsmith Approach

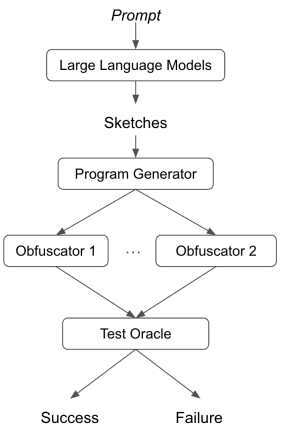

Fig. 2. OBsmith overall workflow

In this section, we present OBsmith, an automated framework that leveraging LLMs to generate sketches and use skteches to test JavaScript obfuscators. OBsmith constructs a multi-step pipeline to test JavaScript obfuscators, its overall workflow is shown in Figure 2.

The first step is **LLM-powered sketch generation** (§3.2 and §3.3). OBsmith contains an initial prompt to define the sketch syntax and instruct LLMs to produce diverse and representative

JavaScript sketches. OBsmith also contains a feedback loop to help LLM generate high quality sketches. Moreover, OBsmith has a extraction framework (§3.4) to automatically extract sketches from existing JavaScript programs, which makes it more powerful.

Then LLM-generated and extracted sketches are used in **program generation** (§3.5) to get concrete JavaScript programs. In program generation stage, OBsmith *(i)* solves the sketch by filling all placeholders and generate concrete expressions in the sketch; and *(ii)* enhances the filled program to log the program's control flow and data flow. The output of program generation stage is a set of enhanced candidate programs ready for obfuscation.

The last step is use candidate programs and call obfuscators to get multiple obfuscated variants. These variants with corresponding ground truth are input of the follow up testing stage. OBsmith contains two test oracles: **differential testing** (§3.6) and **metamorphic testing** (§3.7). OBsmith uses different JavaScript obfuscators with different obfuscation configurations to obfuscate candidate programs. The original candidate program and all obfuscated versions are executed within a standard JavaScript engine (Node.js). Unlike traditional differential testing in compiler testing which lacks of ground truth, OBsmith uses the original program as ground truth. OBsmith collects the execution outputs of all obfuscated programs and compare these outputs with the ground truth. If OBsmith finds any output inconsistencies, it reports the obfuscator as failed. How to define output inconsistencies is in §3.6.

3.1 Sketch Definition

Sketch, also known as "template" or "skeletal program", has proved to be effective in compiler testing and in different programming languages [86, 88]. Previous works define many Domain Specific Languages (DSLs) to represent different sketch syntax, but overall sketch represent a blueprint from which a multitude of concrete, executable programs can be systematically generated. An example of OBsmith's sketch is shown in §2. OBsmith's sketch is an abstract template for JavaScript programs, and we provide a powerful sketch syntax that supports abstractions for numbers, booleans, references, and expressions.

Sketch Syntax. This section provides a formal description of the sketch syntax, and its role within the program generation pipeline (filling and enhancement) that produces the final candidate programs used for differential testing of JavaScript Obfuscators. The primary goal of our sketching language is to enable the test program generation to be both diverse and controlled.

(1) Diversity: The syntax allows programs generation with varied control flows, data flows, and uses of language features. We can ensure the diversity by using same sketch with randomly fill placeholders. This can leads to different control flow and data flow.

(2) Control: Unlike purely random program generation, sketching allows the developers to define a specific program structure or to target a particular language feature, which is useful to inject their domain knowledge. This is essential for creating effective test cases that can isolate and identify bugs related to specific semantic constructs.

By design, our sketch is a syntactically valid JavaScript file where the placeholders are treated as if they were standard identifiers, literals, or expressions. This ensures that the sketch itself can be parsed by standard JavaScript tools (e.g. Babel), simplifying the program generation implementation.

Literal placeholders. are syntactic constructs that represent a location in the code where a new, randomly generated primitive value will be inserted by the filling script. OBsmith supports number and boolean literals now.

A **numberLiteral** represents a placeholder for a concrete numerical value. It can be float or integer (in JavaScript, the only data type is Number, no distinction between integers and floating-point numbers) and replaced with a randomly selected integer or floating-point number (e.g., 10,

-42, 3.14159) in the following fill process. So it must be used in a context where a numerical value is syntactically valid.

A **booleanLiteral** represents a placeholder for a concrete boolean value. It is replaced with either true or false. Must be used in a boolean context, such as in conditional statements or variable assignments.

Reference placeholders. are used to create data flow dependencies in the generated program. The shim replaces these placeholders with references to existing variables of the corresponding type in the current scope. If no such variable exists in the current scope, the generator will throw an error to ensure semantic validity. OBsmith supports both number and boolean references.

A **numberReference** represents a reference placeholder that points to an existing numeric variable in scope. It is replaced by the identifier name of a numeric variable. It is used to create a data dependency on a previously defined numeric variable.

A **booleanReference** represents a reference placeholder that points to an existing Boolean variable in scope. It is replaced by the identifier name of a Boolean variable. It is used to create a data dependency on a previously defined Boolean variable.

Expression placeholders. are used to represent a concrete logic, arithmetic or relation expression. The general expression placeholder is in the following form

```
expression_type(operand1, operand2, operator1, [operator2, ...])
```

For expression type,OBsmith supports arithmetic, relation, and logic. An **arithmetic expression** generates a binary arithmetic expression. The first two parameters represent the operands of the expression. The expression must resolve to a numeric value to be semantically valid at runtime. The following op1, [op2, ...] can be one or more possible arithmetic operators. The generator randomly selects an operator from this set. A **relation expression** generates a binary relational expression that evaluates to a Boolean value. This is the primary construct for creating dynamic control flow. Operand 1 and operand 2 also represent the operands to be compared. They can be of any type. The following op1, [op2, ...] can be one or more possible relational operators. The generator randomly selects one. A **logic expression** generates a binary logical expression. Operand 1 and operand 2 represent the operands of the logical operation. These operands should resolve to Boolean values. The following op1, [op2, ...] can be one or more logical operators.

Operands can be specified in four ways:

- **Placeholders**: OBsmith placeholders for random values (e.g., numberLiteral, booleanLiteral) or random references (numberReference, booleanReference).
- **Identifiers**: Direct references to existing variables in scope (e.g., score, isReady).
- **Literals**: Any literal, such as a string, number, or boolean.
- **Nested Expressions**: An operand can be the result of another expression generator call. This allows us to construct arbitrarily complex and deeply nested program logic from a readable, functional representation. The program generator recursively parses these expressions from the inside out.

3.2 LLM-powered Sketch Generation

OBsmith generates *sketches*—i.e., program templates with typed placeholders that capture domain knowledge about JavaScript—by prompting an LLM with a constrained template language and explicit formatting requirements (Figure 3). The goal is to obtain (i) *diverse* yet *valid* templates that exercise representative control-flow and data-flow patterns, and (ii) sketches that can be deterministically instantiated by our program generator in §3.5.

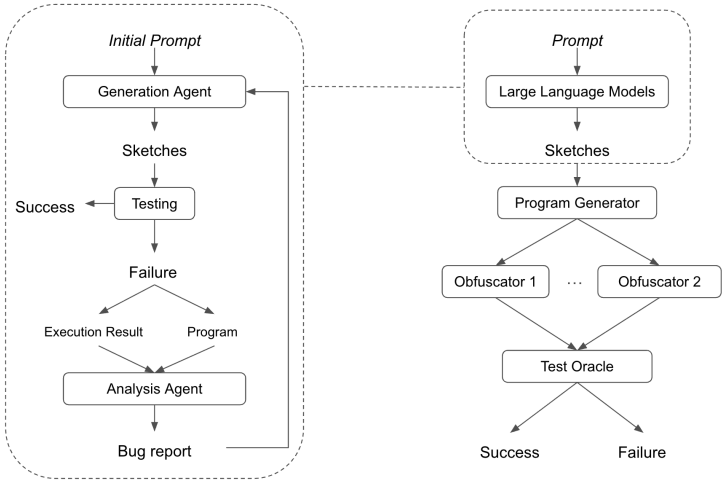

Fig. 3. LLM-powered sketch generation with feedback loop (left) and corresponding location in OBsmith framework

Motivation. LLMs encode rich statistical priors over real-world code and idioms. By coupling a *constrained* prompt with *typed placeholders*, OBsmith harnesses these priors to synthesize compact, human-like templates that systematically expand into large, varied test suites—achieving breadth unattainable with purely random grammars while keeping generator complexity low. Empirically, this design yields high rates of syntactic validity and exposes correctness bugs across obfuscators and configurations (see §4).

Prompt design. The prompt (Figure 5 in appendix) specifies: (1) the sketching **DSL** (literal, reference and expression placeholders); (2) **usage constraints** (operands may be literals, references, or nested expressions); (3) an **I/O format** that yields exactly ten sketches in a fenced code block; and (4) a **worked example** that pairs a template with a filled instance to ground the LLM's understanding of intended semantics. This mix of schema, constraints, and exemplars reduces drifting and encourages syntactically uniform outputs that downstream tooling can parse.

Diversity controls. To broaden coverage, the prompt explicitly requests common JavaScript idioms and control-flow constructs (loops, branches, function calls, array operations), while leaving randomness via placeholders. OBsmith further promotes heterogeneity through: (1) *multi-sketch requests* (ten distinct templates per invocation); (2) *operator sets* in expression placeholders (e.g., `arithmetic(..., +, -, *, /)`), from which the filler later samples; and (3) *nesting* of expression generators to produce deeper ASTs. Together, these encourage varied CFG/DFG shapes from a single, concise specification.

Validity guards in the prompt. Because program generation (§3.5) assumes sketches are *parsable JavaScript with placeholders*, the prompt: (1) requires placeholders to appear where the resulting types are admissible (e.g., `numberLiteral` in numeric contexts, `booleanReference` in predicates); and (2) constrains expression factories to binary operators (logical, relational, arithmetic). These instructions align the LLM outputs with the OBsmith sketch syntax and reduce post-hoc repair.

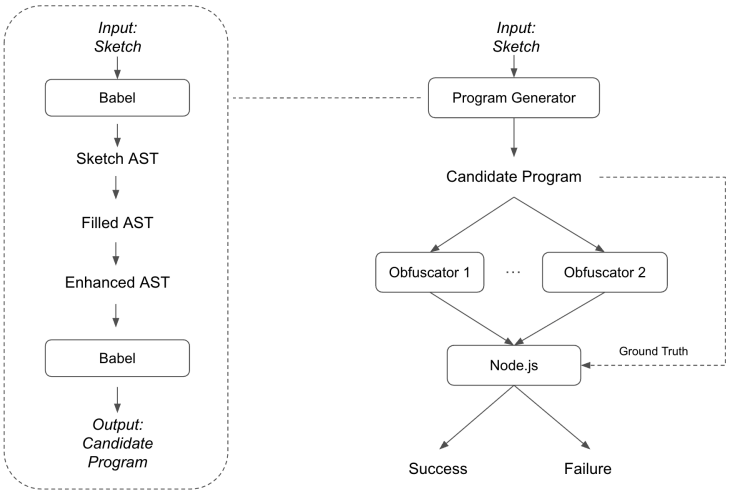

Fig. 4. Program generation workflow (left) and differential testing with ground truth workflow (right)

3.3 Feedback Loop for Sketch Generation

To enhance the effectiveness of LLM-powered sketch generation, OBsmith incorporates a *feedback loop* that automatically leverages prior test outcomes to guide the synthesis of new sketches. This mechanism transforms the testing process from a one-shot generation exercise into an adaptive, iterative cycle that continually improves coverage of bug-revealing scenarios.

Motivation. While LLMs can produce diverse and syntactically valid sketches from carefully designed prompts (§3.2), their output may drift toward common or uninformative patterns. At the same time, differential testing (§3.4) often produces valuable signals in the form of failures, inconsistencies, or execution traces that highlight semantic corner cases. The feedback loop is designed to exploit these signals: instead of discarding failures after bug discovery, OBsmith repurposes them to guide future sketch generation.

Agent Loop Architecture. The feedback loop of OBsmith employs a multi-agent framework with distinct roles:

(1) **Bug report analysis.** A LLM-based bug analysis agent is prompted to analyze the execution result and corresponding failing programs. It summarizes the discrepancy (e.g., suppressed exceptions, scope violations, control-flow divergence) in the structured execution result. Then it generate a *bug report* captures key features of the bug-triggering scenario while abstracting away irrelevant details.
(2) **Sketch refinement.** The second LLM takes the bug report as input and generates new sketches that are likely to reproduce or generalize the observed issue. For example, if the bug report highlights mishandling of `eval()` or incorrect scoping of class constructors, the LLM is instructed to embed placeholders involving these constructs into new sketches.

Interaction with follow up testing. The feedback loop works synergistically with differential testing (§3.6) and metamorphic testing (§3.7). While testing reveals reusable patterns from failing programs, the feedback loop ensures that LLMs actively explore related semantic neighborhoods. This combined mechanism balances data-driven generalization (via LLM) and deterministic reuse (via testing), enabling both novelty and stability in the evolving sketch corpus.

Benefits. By coupling bug-driven feedback with generative modeling, OBsmith can progressively concentrate its testing effort on areas where obfuscators are fragile. This adaptive process reduces redundant sketch generation, improves the precision of bug discovery, and ensures sustained effectiveness as new obfuscators or configurations are introduced. In practice, our evaluation (§4) shows that the feedback loop substantially increases the number of distinct correctness bugs detected compared to generation without feedback.

Summary. The feedback loop elevates OBsmith from a static testing framework to a self-improving system. By turning runtime failures into structured prompts for sketch synthesis, it creates a virtuous cycle in which every discovered bug enriches the generator, leading to more robust and targeted testing in subsequent iterations.

3.4 Sketch Extraction from Real-World Program

In addition to LLM-powered sketch generation (§3.2), another component of OBsmith's sketch generation is automatic sketch extraction, which aims to automatically extract reusable program sketches from real-world JavaScript programs.

Motivation. Differential testing and metamorphic testing (§3.6, §3.7) frequently reveal program fragments that expose semantic inconsistencies in obfuscators. These fragments often encode subtle interactions between JavaScript features—such as scoping rules, exception propagation, or dynamic evaluation—that are difficult to anticipate in manual prompt design. Automatically converting such fragments into parameterized sketches increases test diversity while focusing future iterations on bug-revealing structures. Moreover, sketch extraction can automatically use existing JavaScript programs, which usually contain the developer's domain knowledge and have been proven to be effective for compiler testing[85]. For the above two reasons, OBsmith also designed an automated framework to extract sketches from existing programs.

Extraction Pipeline. OBsmith implements the extraction in AST level, it parses the concreate JavaScript program to a Babel AST and proceeds the sketch extraction in two steps:

Step 1: **Collect candidates.** Identify variables that hold numbers or booleans.
Step 2: **Replacement.** The program is parsed into an Abstract Syntax Tree (AST), and literals, variable references, and selected expressions are abstracted back into OBsmith placeholders (`numberLiteral`, `booleanReference`, `arithmetic()`, etc.). The generalization step ensures that the resulting sketch is type-correct and can be re-instantiated with diverse values.expressions

Benefits. By automating the discovery of new sketches, OBsmith transforms one-off bug-triggering programs into systematic generators of test cases. This mechanism reduces manual effort, broadens semantic coverage, and enables continuous adaptation of the testing pipeline as new obfuscators or language features emerge. Importantly, sketch extraction allows OBsmith to evolve its corpus in tandem with the domain, turning transient failures into long-lasting assets for regression testing.

Summary. Together with LLM-powered sketch generation, program filling, and differential testing, sketch extraction completes a feedback-driven workflow that continually enriches OBsmith's test suite. This synergy ensures that OBsmith is not only capable of finding bugs once, but also of systematically amplifying its effectiveness over time by reusing and refining bug-inducing patterns.

3.5 Sketch-based Program Generation

OBsmith generate sketches using LLMs and auto-extraction, these sketches are input to the program generator. Program generation is a two-step process that transforms partial sketches into complete, executable JavaScript programs which are ready for differential and metamorphic testing.

Step 1: *Sketch filling*. OBsmith systematically replaces all placeholders and incomplete expressions in each sketch. OBsmith traverses the sketch and substitutes each placeholder with a concrete value or expression, producing a filled concrete program.

Step 2: *Program enhancing*. Filled programs are augmented with testing oracles that enable straightforward equivalence checking between original and obfuscated code. These oracles ensure that behavioral differences can be reliably observed during execution. The output of this stage is a set of candidate programs, ready to be obfuscated and evaluated.

3.5.1 Sketch filling. The sketch solving is a specialized code transformation phase to transform an abstract sketch into multiple, unique, and concrete JavaScript programs. It systematically fills the literal and reference placeholders with random value or variables and uses an "expression generator" to fill expression placeholders within a sketch, effectively translating the high-level program template into executable JavaScript code. This process is the first and critical step in creating the diverse corpus of programs required for effective differential testing.

We implement sketch filling algorithm in Babel, which allows parsing with placeholders into an AST. For each program to be generated, the sketch is parsed into an AST using a standard Babel JavaScript parser. OBsmith traverses and fills placeholders using a visitor pattern. At each node, it checks for the presence of OBsmith's specialized placeholders. When a match is found, it performs a transformation. OBsmith's sketch filling logic is divided based on the type of placeholders encountered during the AST traversal.

Literal placeholders. (numberLiteral and booleanLiteral). These are the simplest transformations, typically involving the replacement of an Identifier node in the AST. (numberLiteral, booleanLiteral): When an identifier matching one of these placeholders is found, it is replaced with a new Literal node in the AST. The value of this literal is a randomly generated primitive of the corresponding type. For booleanLiteral, OBsmith simply replace it with a True or a False with equal probability. Since JavaScript only provides the Number class and doesn't distinguish between Integer and Float numbers, for numberLiteral, OBsmith generates an random integer or a random float with equal probability.

Reference placeholders. (numberReference and booleanReference). This is a more complex operation that requires scope analysis. When a reference placeholder is found, the script traverses up the AST from the current node to find all variable declarations in the current and parent scopes. It filters this list to find variables of the correct type and randomly selects one from the filtered results. The placeholder Identifier node is then replaced with a new Identifier node containing the name of the selected variable. If no suitable variable is found in current scope, OBsmith uses a corresponding boolean or number value to fill the sketch.

It's worth noting that in addition to using explicitly defined variables (`let a = 1; let b = True`), OBsmith also supports filling sketches with elements from arrays. Unlike other programming languages that contains strict type check mechanism, JavaScript allows elements of different types to exist in the same array. For example, `let arr = [True, 3.1415926, "Hello World!", 1];` is a valid statement in JavaScript. If `arr` is accessible in current scope, OBsmith can use arr[0] to fill booleanReference and use arr[1] or arr[3] to fill numberReference.

Expression placeholders. (e.g. arithmatic(operand1, operand2, operator1, operator2 ...)) This is the most sophisticated part of the sketch filling algorithm. The expression placeholders are identified as CallExpression nodes in Babel AST. When a CallExpression node whose callee is one of the generators (arithmetic, relation, logic) is found, the script performs the following steps:

Step 1: *Argument Resolution:* The first two parameters represent the left and right operands, respectively. OBsmith recursively processes these two parameters passed to the generator function. If operand is another expression placeholder, the inner expression placeholder is parsed first. If it is a literal or reference placeholder, it is replaced as described above. This recursive population method correctly constructs deeply nested expression placeholders.

Step 2: *Operator Selection:* Starting with the third parameter, the operators that can be used for this expression placeholder are represented. OBsmith examines the operator parameter (which can be a string literal such as "+", "*", "===", or the operator itself +, *, ===). OBsmith randomly selects one of these operators to use in the final expression.

Step 3: *Node Transformation:* The original CallExpression node is removed from the AST and replaced with a new BinaryExpression node. This new node's left and right attributes (operand) are set to the parsed operand parameters, and its operator attribute is set to the randomly selected operator.

Invalid expression syntax. Although we have instructed LLMs to generate valid sketches through a series of prompting techniques as shown in §3.2, we still find that LLM generate syntactically incorrect sketches that are different from our intention. To enhance usability, OBsmith has designed error tolerance mechanisms for some common error types to reduce program generation failures caused by operator errors. Specifically, the following measures are taken for common operator errors.

Error 1: *Use unary operator in expression placeholder.* If a unary operator *unary_op* appears in an expression placeholder and *unary_op* is selected when filling sketch, OBsmith generates a unary expression node using the first operand and ignoring the second operand. The unary expression node will replace the original expression placeholder.

Error 2: *No operator in expression placeholder.* If there is no operator in the expression placeholder, and only two parameters represent the left and right operands, OBsmith will randomly select one from all Binary operators and use these two parameters as the left and right operands to generate a binary expression node, which will replace the original expression placeholder.

After the AST traversal and filling are complete for one iteration, the modified AST—which no longer contains any OBsmith defined placeholders—is passed to Babel, which converts the AST back into a syntactically correct JavaScript program. The process is repeated multiple times to produce a set of distinct programs from a single sketch, with randomness introduced at specific, controlled placeholders.

3.5.2 Program Enhancing. Program enhancement is a source-to-source transformation that takes a filled JavaScript program (in §3.5.1)as input and systematically injects logging code to produce an enhanced program. The goal is to make the program's execution process observable and deterministic but not change it's behavior and functionality. OBsmith implements the program enhancement in Babel. The input (filled) program is first parsed into an AST, which OBsmith traverses top-down using a visitor pattern. During traversal, new `ExpressionStatement` nodes are injected at locations such as `BlockStatement` or `FunctionDeclaration`, with each `ExpressionStatement` node containing a `CallExpression` (e.g., `console.log`) for logging. After traversal, the enhanced AST is converted back to JavaScript code and written to an output file. The enhancement involves several different types of logging code injection, each

designed to provide different aspects of runtime observability. Specifically, we inject the following logging instruments.

Global error handling. JavaScript is a lightweight, function-first, interpreted (or just-in-time compiled) language that executes code line by line. Because of this characteristic, JavaScript lacks a main function, making it difficult to determine the complete execution of a program. Therefore, we designed global error handling to wrap the entire program body in a global try...catch block. Specifically, the program's top-level statements are moved into the body of a TryStatement node. A CatchClause is added to handle any exceptions, logging the caught error and the error message. This ensures that any uncaught exceptions during program execution are caught and logged in a standard format (e.g., "!!! GLOBAL ERROR Caught!!!"). When an error is caught, this mechanism also forces the program to exit with a non-zero status code (process.exit(1)), clearly signaling failure to the test script.

Block level control flow tracing. OBsmith identifies every block statement in the AST. A block statement is a sequence of statements enclosed in curly braces, such as a function body in function declaration node, the consequent and alternate block of an if statement, or the body of a for or while loop. OBsmith contains both block entry and block exiting log. Specifically, a console.log statement is prepended to the beginning of every block. This log indicates entry into the block, using its location in the source file as a unique identifier (e.g., "-> Entering Block@<line>:<col>"). A console.log statement is appended to the end of every block to signal its completion (e.g., "<- Exiting Block@<line>:<col>").

Data flow tracing via state checksum. This is a critical instrumentation for detecting data-flow bugs. Just before the exit log of every block, another console.log is injected. This statement performs scope analysis to identify all accessible variables from current block. It then logs the name and current value of each variable. The Log Format is as follows " — Checksum for Block@<line>:<col> — var1: value1, var2: value2, ..." This "checksum" provides a snapshot of the program's data state at the end of every basic block. OBsmith uses this output to verify that the data state of an obfuscated program matches the baseline at every step of execution.

Function call instruments. To observe the arguments passed to functions, additional logging is injected. At the beginning of every function's body (immediately after the block entry log), a console.log is added. The Log Format is "=> Entering function: <functionName> Arguments: <arg1>, <arg2>, ..." This log makes the dynamic call graph and the data passed between functions explicit, allowing for the detection of bugs where an obfuscator might reorder or incorrectly modify function arguments.

3.6 Differential Testing

Differential testing is a widely adopted technique in compiler validation [80, 86], where the absence of a reliable ground truth necessitates comparing the outputs of multiple compiler variants on the same input program. In contrast, OBsmith operates under the assumption that the original JavaScript program, executed directly in a trusted engine (Node.js), serves as the definitive ground truth. This allows OBsmith to adapt differential testing specifically for the obfuscation domain.

Methodology. Given a candidate program generated from a sketch, OBsmith applies multiple obfuscators (e.g., JS-Confuser, Obfuscator.IO) under different configuration settings. The original and obfuscated programs are then executed within Node.js. OBsmith collects and normalizes their execution outputs, where the output of the original program is treated as the reference oracle.

Equivalence Checking. For each obfuscated variant, OBsmith compares its execution results with the oracle. The comparison includes:

- **Standard Output:** Console outputs must be identical across all runs.
- **Exceptions:** Exception types and messages must match; stack traces are ignored due to nondeterministic file names and line numbers.
- **Termination Behavior:** Programs must terminate consistently. Divergences such as premature exits or runtime crashes are flagged as failures.

A mismatch in any category constitutes a correctness bug introduced by the obfuscator.

Timeout Handling. Randomly generated candidate programs may introduce non-terminating behavior (e.g., `while(true)`). To avoid indefinite execution, OBsmith enforces a strict timeout of 60 seconds. If both the original and obfuscated programs exceed this limit, the test case is considered consistent and marked as a pass. Otherwise, the discrepancy is recorded as a failure.

Summary. By leveraging the original program as an oracle, OBsmith strengthens differential testing with a reliable ground truth. This adaptation enables precise detection of semantic inconsistencies introduced by obfuscation, distinguishing it from traditional approaches that rely on consensus among potentially flawed variants.

3.7 Metamorphic Testing

Differential testing (§3.6) is our primary mechanism for checking semantic equivalence between an original program and its obfuscated counterpart. In addition, we explore whether *metamorphic testing* can stress obfuscators by applying equivalence-preserving rewrites to the *inputs* before obfuscation and then verifying that obfuscation does not change program behavior. The goal here is not limited to the "single-obfuscator available" setting; rather, it is to probe whether obfuscators are robust to semantics-preserving variation in their inputs.

Motivation. Many source-level rewrites leave a program's observable semantics unchanged. If an obfuscator is semantics-preserving, then applying such a rewrite to a program *before* obfuscation should not alter the behavior of the resulting obfuscated code. For example, constant folding, algebraic rewriting, or renaming of variables should not alter the runtime behavior of a program. If applying such a semantics-preserving transformation before and after obfuscation produces divergent outputs, the obfuscator has introduced a correctness bug.

Metamorphic Relations. OBsmith defines a suite of metamorphic relations (MRs) over JavaScript programs, our implementation use three most frequently used MRs in compiler testing:

- **Algebraic equivalence:** Replacing $x + 0$ with x, $x \times 1$ with x, or $(x + y) - y$ with x.
- **Control-flow equivalence:** Transforming `if (true) {S}` into S, or flattening nested conditionals into equivalent disjunctions.
- **Dead-code injection/removal:** Adding or removing statements that provably do not affect program outputs (e.g., `if (false) {...}`).

Each MR captures a known semantic invariant. The transformations are automatically applied to programs generated from sketches, and both the original and transformed variants (by MRs) are processed by the obfuscator under test.

Testing Workflow. For each sketch instantiation:

(1) Generate a concrete program P and one or more metamorphic variants $\{P_1, P_2, \dots\}$ using the MRs above.

(2) For each obfuscator O under test, apply O with different configuration settings to P and to each P_i, producing multiple obfuscated versions $\{O(P), O(P_1), \dots\}$.
(3) Execute all obfuscated programs under the same runtime environment, collecting their outputs and observable side effects.
(4) Report a correctness bug if any obfuscated version of P and its corresponding P_i differ in output, runtime exception behavior, or termination status.

Benefits. Metamorphic testing directly evaluates an obfuscator's *invariance* to semantics-preserving rewrites, exposing order-sensitivity and brittle interactions between transformation passes (e.g., encoder/decoder pipelines, control-flow flatteners, scope virtualizers). It is equally applicable when multiple obfuscators are available and when only one is present, and it supports lightweight regression tests across versions/configurations of the same tool.

Summary. By encoding program equivalences as metamorphic relations, OBsmith provides a lightweight yet powerful mechanism to validate obfuscators in the absence of oracles. This technique, in combination with differential testing, yields comprehensive correctness checking across both multi-obfuscator and single-obfuscator scenarios.

4 Results

To evaluate OBsmith, we study the following research questions:

- **RQ1:** How well LLMs follows the prompt and generate syntactically valid sketches?
- **RQ2:** What critical bugs does OBsmith detect in real-world JavaScript obfuscators?
- **RQ3:** How effective is OBsmith compared with the state-of-the-art techniques?
- **RQ4:** What are the contributions of the major components of OBsmith?

4.1 Experimental Setup

4.1.1 Large language models. Regarding the choice of LLM, we chose the most powerful version of the LLM model currently on the market. For each llm, we generate 100 different sketches. We also test the coding agent and see its effectiveness in generating sketchs compared to directly use base model. For fair comparison since coding agent like Gemini-CLI or Claude Code can see the whole project, we generate sketches first and then put other LLMs results into our project directory. Specifically, we use the following models: Gemini 2.5 Pro, GPT-5, Claude 4 Opus, Qwen3-235B-A22B-2507, Grok 4. Besides, we also evaluate LLM agent's ability in sketch generation, we use Gemini CLI with Gemini 2.5 Pro to generate 100 sketches as well. In order to fair comparison, we begin with Gemini CLI without other model's result, in other word, we do this first.

4.1.2 Obfuscators under test. Based on Google's investigation[30], Obfuscator.IO[4] and JS-Confuser are two most wide used JavaScript obfuscator in Google's production. So our main targets are the two most popular JavaScript obfuscators: JS-Confuser and Obfuscator.IO. JavaScript obfuscation takes candidate program as input alongside a JSON configuration.

Recognizing that the behavior of an obfuscator can vary significantly based on its settings, we systematically applies different configurations to each candidate program. This results in the generation of multiple, distinctively obfuscated versions of each original program, thereby maximizing the coverage of the obfuscator's feature set and potential transformation space. JS-Confuser have 3 preset configurations - low, medium and high. We didn't make any changes to its configurations. Obfuscator.IO have 4 preset configurations - Default, Low, Medium, and High. We use these preset.

However, our OBsmith implementation use console log to check program equivalence, but in the low, medium, and high settings, the console was disabled so we change the option to false. For high

level, Obfuscator.IO enable debug protection which will make the debug tools useless but make the script hangs forever, so we also set that option to false. We didn't make any other changes on configurations. Compact (put all things in one line) is also disabled in setting since V8 will reject code with too long line trigger a javascript invalid size error.

4.1.3 JavaScript Engine. We use Node.js to run our experiments. At its core, Node.js is a server-side JavaScript runtime environment built on the foundation of the V8 JavaScript engine. V8 is developed by Google and used in its Chrome browser, which is responsible for taking the JavaScript code written in a Node.js application and compiling it into the machine code that the computer can execute directly. This compilation process, leveraging V8's Just-In-Time (JIT) compilation and optimization techniques, allows Node.js applications to achieve high levels of performance and efficiency, particularly in handling concurrent connections and asynchronous operations, says GeeksforGeeks. Essentially, V8 provides the raw power of JavaScript execution, while Node.js extends that power with a rich set of APIs and modules, making it a complete environment for building scalable server-side applications

4.1.4 Extraction dataset. To evaluate sketch extraction, we uniformly sampled 1,000 source files from the 150k JavaScript Dataset [56]. For each file, our extractor produced a single sketch. We then instantiated every sketch into three concrete programs, yielding a total of 3,000 instantiated programs for downstream evaluation.

4.1.5 Baseline. We select five JavaScript fuzzers (FuzzJIT[74], Jsfunfuzz[2], DIE[55], Fuzzilli[23], and Superion[73]) as baselines. FuzzJIT[74] is an oracle-enhanced fuzzing technique to detect non-crashing and crashing JIT compiler bugs. Jsfunfuzz is a generation based fuzzer for JavaScript/Node packages (practical for fuzzing JS libraries). DIE advances the crossover by restricting the types of sub-tree. Fuzzilli is a coverage-guided fuzzer for JavaScript engines based on a custom intermediate language, FuzzIL, which can be mutated and translated to JavaScript. Instead of mutating the AST, or other syntactic elements of a program, FuzzIL facilitates convenient mutations on the control and data flow of a program. A FuzzIL program contains a list of instructions, and can be lifted to a JavaScript program for testing. Fuzzilli is popularly adapted to build powerful JavaScript fuzzer by academia researchers and industry practitioners. Superion finds bugs by conducting crossover on the AST sub-trees of two parent samples.

In our comparison experiments, we use the implementations of baselines except FuzzJIT provided by UniFuzz[38], which is a fuzzing approach evaluation benchmark and provides a collection of docker for 37 well-known fuzzers, including Superion, DIE, Jsfunfuzz, and Fuzzilli, to ease the evaluation of different fuzzing approaches. Only Superion and DIE require initial seeds. DIE's initial seed corpus contains 100 JavaScript files generated by its given script, and the same set of initial seeds are also used for Superion. For FuzzJIT, we use their official implementation.

4.2 RQ1: LLMs' Sketch Generation Ability

Table 1 summarizes the quality of sketches produced by different LLMs. We evaluate (i) syntactic validity of generated sketches and (ii) adherence to the sketch spec, focusing on five recurring error modes: No-op (missing operator in a composite expression, especially in nested expressions), Ternary op (using ternary where our DSL forbids it), Unary op (introducing unary ops in expressions), Placeholder (using unsupported placeholder types), and Self-defined op (invented operators not in our grammar).

Overall performance. The strongest models—GPT-5 (98% valid), Claude 4 (96%), and Gemini 2.5 Pro (94%)—largely followed the prompt and DSL. Their residual errors were rare and concentrated in operator handling. Our error-tolerance pass converts sketches with a missing operator or stray

Table 1. Syntax validity and error types in LLM-generated sketches, each model generate 100 sketches

	Valid Syntax	No op	Ternary op	Unary op	placeholder	Self-defined op
Gemini 2.5 Pro	92	×		×		×
GPT 5	98			×		
Claude 4	96		×	×		
Qwen 3	88	×	×	×		
Grok 4	69	×		×	×	×
Gemini CLI	94	×		×		

unary operator into runnable programs, preventing pipeline crashes. *Ternary op*, *Unsupported Placeholder*, and *Self-defined op* violations are unrecoverable within our DSL.

Per-model breakdown.

- **GPT-5 (98% valid).** Almost fully compliant; we observed no systematic category of spec violation in the sampled set. Occasional minor formatting irregularities were corrected by the sanitizer.
- **Claude 4 (96% valid).** High compliance overall, with a small number of *ternary* insertions (e.g., embedding `cond ? a : b` inside a placeholder). These violate our sketch grammar.
- **Gemini 2.5 Pro (92% valid).** Solid adherence with two recurring operator issues: (i) *No-op*—omitting a logical or arithmetic operator inside a nested expression; and (ii) *Unary op*—placing `!` or `++` inside placeholders. Both are handled by our tolerance mechanism (assigning a safe default operator; stripping/hoisting the unary op).
- **Qwen 3 (88% valid).** Partial task understanding but frequent violations of our constraints. The dominant errors are *No-op* omissions and *Ternary op* use inside placeholders, which cannot be auto-repaired and would otherwise crash sketch filling.
- **Grok 4 (69% valid).** Most non-compliant. In addition to *No-op* and *Unary op* misuse, Grok frequently introduced *Placeholder* types outside our DSL (strings) and Grok is the only model to fail in this area. Same as Qwen, Grok also *Self-defined op* which is out of our sketch syntax.
- **Gemini CLI (94% valid).** Using the same base model as Gemini 2.5 Pro but with repository context, the agent followed the sketch–fill–enhance workflow reliably. The main residual issue was occasional *Unary op* and *No-op* omissions.

Code-agent vs. base model. The Gemini CLI agent (94% valid) shows that lightweight code awareness—file-system context plus access to repository scripts and documentation—substantially reduces spec violations relative to its base model. Residual issues are mostly occasional missing-operator ("no-op") cases. Importantly, all Gemini CLI errors are automatically recoverable by our pre-processing/filling scripts, likely because the agent adheres more closely to the sketch–fill–enhance interfaces exposed by the repository.

Answer to RQ1: LLMs' Sketch Generation Ability

GPT-5, Claude 4, and Gemini 2.5 Pro generate sketches that are both syntactically valid and DSL-conformant in the vast majority of cases, with residual issues concentrated in operator specification that our tolerance pass can absorb. Qwen 3 and Grok 4 struggle with domain-specific constraints, producing unrecoverable DSL violations. Providing task context via a code agent improves adherence without changing the underlying LLM.

4.3 RQ2: Case Study

Using OBsmith, we systematically reveal correctness bugs in both Obfuscator.IO and JS-Confuser. All identified bugs were reproducible and confirmed across both the online version in the obfuscator's website and the latest releases available in their respective GitHub repositories.

4.3.1 *Bugs in JS-Confuser.*

- **Constructor name corruption**: JS-Confuser alters the constructor name of classes. For example, evaluating `console.log(bx.constructor.name)` produces incorrect results across all three configuration levels.
- **eval() crashes**: Programs that execute correctly in their original form crash after obfuscation, consistently across all configurations.
- **Silent "fixes" of exceptions**: In medium and high configurations, JS-Confuser modifies behavior so that programs that should raise exceptions instead complete execution without error.
- **Incorrect error handling**: In medium and high configurations, expected exceptions are either suppressed or altered, resulting in the loss of valuable debugging information.
- **Scope violations**: Variables accessed out of scope return `undefined` post-obfuscation, whereas debug information was available beforehand.
- **Control-flow modifications**: In some medium-level configurations, the control flow is altered, leading to different return values. For instance, programs that should terminate with an exception instead end normally.
- **Undefined value mishandling**: Similar to the control-flow issues, `undefined` values are silently "corrected," altering return types. This occurs in medium and high configurations, effectively masking underlying errors.

4.3.2 *Bugs in Obfuscator.IO.*

- **Unhandled constructs**: Certain valid JavaScript constructs cannot be obfuscated. Examples include:
  - `async function () { } / 1;`
  - `class x { static { function x() { } function x() { } } }`
  - `! class { }();`
- **Type errors**: Some transformed programs result in runtime `TypeError`s, breaking semantic equivalence with the original code.

Answer to RQ2: Case Study

These findings demonstrate that correctness bugs are pervasive in widely used obfuscators. While some errors manifest as program crashes, others are more insidious—such as silent changes in return values or suppressed exceptions—that can undermine program reliability and complicate debugging. OBsmith's LLM-driven, sketch-based differential testing proves highly effective at surfacing such issues, reinforcing the need for rigorous correctness validation in obfuscator development.

4.4 RQ3: Comparison Experiment

We compare OBsmith against five JavaScript fuzzers (FuzzJIT, JSFuzz, Superion, DIE, and Fuzzilli) on the same backend (V8 as shipped in Node.js) under an equal program budget. Concretely, we use six generators—five LLMs plus one agent (Gemini CLI) —to produce **600** sketches (100 per LLMs) and instantiate **5** programs per sketch, yielding **3,000** programs. Each technique then exercises

Table 2. Comparison with baseline and ablation study

Bug	Comparison with baselines						Ablation study				
	OBsmith	FuzzJIT	Jsfunfuzz	Superion	DIE	Fuzzilli	-l	-f	-e	-m	-d
constructor name	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
eval crash	✓						✓				✓
silent fix	✓						✓				✓
error handling	✓						✓	✓	✓		✓
scope violation	✓						✓		✓		✓
control flow change	✓	✓	✓				✓	✓	✓		✓
undefined handling	✓						✓	✓			✓
async crash	✓		✓						✓		✓
class crash	✓						✓	✓			✓
! calss crash	✓						✓				✓
type errors	✓						✓				✓

V8 on this budget, and we record whether a technique exposes a unique bug (newly observed misbehavior) under our oracle. Baselines were taken from UniFuzz where available; Superion and DIE were supplied with the required default seed inputs.

Results. Table 2 shows that OBsmith is the *only* technique that exposed the bugs we report under this setting; none of the five fuzzers reproduced these failures within the same execution budget. The correctness regressions we surface (e.g., exception suppression, scope violations, constructor-name corruption, and eval() crashes) were all triggered by programs produced via our sketch–fill–enhance pipeline and were *not* rediscovered by general-purpose engine fuzzers in our runs.

Interpretation. This outcome is expected: the baselines are optimized to find *engine* defects (JIT/VM bugs) via coverage-guided mutation, while OBsmith is specialized for validating *semantics-preserving transformations* by obfuscators, with an oracle that checks output/exception/termination equivalence across original and transformed programs. The comparison therefore demonstrates complementary strengths rather than a head-to-head replacement.

Answer to RQ3: Comparison Experiment

Under an equal program budget (3,000 programs), OBsmith surfaced all observed correctness bugs, whereas state-of-the-art JavaScript fuzzers did not expose these issues in our setting. The evidence supports using OBsmith alongside engine fuzzers: the former targets transformation-induced semantic drift; the latter remains essential for VM/JIT vulnerabilities.

4.5 RQ4: Ablation Study

We evaluate how each OBsmith component contributes to bug discovery by enabling exactly one sketch-generation technique or exactly one oracle at a time: the initial LLM sketcher (**-l**), the feedback loop that refines sketches (**-f**), automatic sketch extraction from real code (**-e**), metamorphic testing (**-m**), and differential testing (**-d**). Every variant is run under the same program budget as

the full system. For the feedback setting, we report only *new* bugs found by the refined sketches—if the input already exhibits "bug i" and the loop merely reconfirms bug i, we do not count it.

Results. Table 2 reports bug classes exposed by each technique. Overall, every component except metamorphic testing contributes unique coverage.

- **Differential testing (-d)** is the single most impactful oracle. It uncovers the majority of classes tied to *semantic divergence* between the original and obfuscated program—e.g., *eval* crashes, incorrect *undefined* handling, *type errors*, and general *error-handling* inconsistencies. These require cross-version comparison and are rarely surfaced by engine fuzzers.
- **LLM-driven generation (-l)** and the **feedback loop (-f)** each expose multiple *API/semantics–sensitive* issues such as corrupted *constructor.name*, *scope violations*, *silent fixes* (exceptions suppressed or behavior altered without a visible crash), *control-flow changes*, and several *class/async* crashes. The loop frequently converts borderline-invalid or underspecified sketches into valid, higher-tension programs that better stress obfuscators.
- **Automatic extraction (-e)** complements learned sketches with patterns mined from real code. Extracted sketches hit bug classes that benefit from idiomatic structures (e.g., class hierarchies and nested control flow), overlapping with but not subsumed by (-l) and (-f).
- **Metamorphic testing (-m)** did not reveal additional bug classes under our three generic compiler MRs. This suggests that *obfuscator testing needs domain-specific MRs* (e.g., invariants about identifier renaming, string-table transformations, or control-flow virtualization) rather than the generic arithmetic/logical equivalences commonly used for compilers.

Answer to RQ4: Ablation Study

(i) Every component except -m contributes to at least one bug class, with **-d** providing the largest unique lift.
(ii) **-l**, **-f**, and **-e** provide complementary coverage: LLM-only sketches (-l) already stress obfuscators; feedback (-f) recovers additional crash types (notably *class*); and extraction (-e) adds realistic patterns that trigger control/data-flow issues and reveal async crash.
(iii) The negative result for **-m** suggests that *general* compiler MRs are ill-suited to obfuscator correctness; domain-specific MRs are likely required to make metamorphic testing effective. However, it specifically surfaces brittleness like order-sensitivity and bad interactions between transformation passes that ordinary differential tests can miss. It also complements the differential oracle as a lightweight regression harness across versions/configurations, broadening coverage without needing new ground-truth outputs.

5 Discussion

Based on investigation from Google[30] and WeChat[39], runtime performance is also important in JavaScript obfuscation, so beyond correctness of obfuscators, we also evaluate the obfuscation effect on storage runtime performance. To evaluate OBsmith's ability to surface these trade-offs, we measured average file size, runtime execution time, and memory usage across multiple obfuscation configurations. Table 3 summarizes these results.

Moderate obfuscation settings revealed by OBsmith show manageable costs. When applying Obfuscator.IO (Default, Low) and JS-Confuser (Low) configurations, OBsmith reported only minor runtime overheads (<3%) and negligible memory differences (<1%), despite moderate file size increases (76-694%). These observations demonstrate that OBsmith can reliably detect small but consistent performance shifts under lighter obfuscation.

Table 3. Performance comparison of obfuscation configurations with absolute values and percentage changes from original program (filled and enhanced).

Configurations	Avg file size (KB)		Run time (ms)		Memory usage (KB)	
	Abs.	%Δ	Abs.	%Δ	Abs.	%Δ
Original program	2.092	0.00%	28.8	0.00%	32,311.39	0.00%
Obfuscator.IO (Default)	3.686	+76.18%	28.8	+0.06%	32,385.93	+0.23%
Obfuscator.IO (Low)	4.267	+103.96%	29.0	+0.46%	32,464.06	+0.47%
Obfuscator.IO (Medium)	15.486	+640.17%	33.5	+16.04%	38,193.16	+18.20%
Obfuscator.IO (High)	37.176	+1,676.89%	44.7	+55.07%	43,835.42	+35.67%
JS-Confuser (Low)	16.621	+694.40%	29.5	+2.42%	32,517.74	+0.64%
JS-Confuser (Medium)	49.097	+2,246.69%	53.6	+85.95%	34,817.05	+7.75%
JS-Confuser (High)	159.207	+7,509.55%	113.9	+295.07%	45,408.50	+40.53%

Aggressive obfuscation highlights OBsmith's sensitivity to extreme overheads. OBsmith also exposed the dramatic costs of higher settings. For instance, Obfuscator.IO High inflated file size by +1,676.89% and runtime by +55.07%, while JS-Confuser High ballooned file size by +7,509.55%, runtime by +295.07%, and memory by +40.53%. Even the Medium settings produced noticeable slowdowns (+16% runtime for Obfuscator.IO, +85.95% for JS-Confuser). These measurements confirm that OBsmith can capture both moderate and extreme performance penalties.

By systematically reporting file size, runtime, and memory metrics across obfuscation configurations, OBsmith provides actionable evidence that stronger obfuscation often comes at the cost of severe inefficiency—insights essential for both researchers and practitioners.

6 Threats to validity

As with any empirical study, OBsmith is subject to internal and external threats.

Internal. (1) OBsmith relies on LLM-generated sketches, which are inherently non-deterministic. Different runs may produce different sketches and uncover different bugs. To mitigate this threat, we use multiple LLMs and generated 100 sketches in each run and report aggregated results.

(2) The equivalence-checking mechanism may miss subtle semantic differences (e.g., performance or memory behavior). Compared to prior work, we strengthen equivalence checking by incorporating both control-flow validation and scope-level checksums. In addition, we execute programs multiple times to reduce nondeterministic noise.

(3) The feedback loop introduces another potential source of bias. Such interaction may reinforce trivial variations instead of producing genuinely novel cases. To control for this, we compared bug-finding effectiveness with and without the feedback loop. Our results show that the loop improves bug discovery.

External. OBsmith execute all programs in Node.js engine, which may not fully represent the behavior of JavaScript engines embedded in browsers or alternative run times. Bug manifestation could vary across environments. In addition, although we evaluated multiple widely used open-source obfuscators, our findings may not generalize to all available or proprietary tools. Nevertheless, the chosen obfuscators are representative of current practice in real-world production (based on Google's investigation), and we believe the results provide a meaningful characterization of the reliability of existing obfuscators.

7 Related Work

7.1 JavaScript Obfuscator

Software obfuscation deliberately transforms code to hinder readability, analysis, and reverse engineering. Common obfuscation techniques include restructuring code, replacing descriptive variable names with unintuitive identifiers, injecting redundant or misleading instructions, manipulating control flow, and encrypting literals such as strings or configuration data[8, 89]. Although obfuscation can legitimately safeguard proprietary algorithms and sensitive resources (e.g., IP addresses)[16, 44], it is also exploited by attackers to mask malicious logic—especially in web and mobile scripts[11, 46, 58, 61].

Obfuscated code severely diminishes the effectiveness of analysis tools, creating a major obstacle for software testing and static analysis methods[7, 78, 87]. Moreover, obfuscation hampers malware detection by concealing malicious behavior from static analyzers, security filters, machine-learning detectors, and manual reviewers[37, 54, 57]. JavaScript, the predominant language for client-side web applications, is especially vulnerable to obfuscation due to its inherently open and accessible nature [10, 13, 20].

Previous studies have explored various dimensions of software obfuscation, encompassing obfuscation techniques, detection methodologies, and evaluation frameworks. Sun et al. [70] highlighted the significant adverse impact of obfuscation on anti-malware tools. Similarly, Hammad et al. [25] conducted extensive experiments to evaluate the efficacy of leading anti-malware products against diverse obfuscation methods, including seven prominent open-source, academic, and commercial obfuscation tools. Furthermore, various approaches have been proposed specifically to detect JavaScript obfuscation.

However, existing research largely overlooks critical aspects such as evaluating whether obfuscation tools inadvertently alter the original code functionality or negatively impact runtime performance. Skolka et al.[65] studying minified and obfuscated code in the web, however, it study some obsolete test cases and JavaScript TypeScript changes a lot in recent years the thoughts are unavailable now. E.g. Based on Google's investigation[30], Obfuscator.IO is the most frequently used obfuscator now 90% of obfuscated code was obfuscated by this, but Skolka's work didn't cover this work in total. So it's time to rethink this.

7.2 Testing and Sketching.

Differential testing[43, 48, 62, 72, 80, 82, 84, 85] is a software testing technique in which multiple implementations of the same specification are tested using the same set of inputs, with discrepancies in their outputs serving as indicators of potential defects. Csmith[80] is a well-known tool for testing C compilers by randomly generating C programs. It found bugs in main-stream compilers and led to significant attention for compiler testing. Recently, injecting domain knowledge and real-world code has been shown to be effective in compiler testing. Skeletal program enumeration (SPE)[88] technique to test C/C++ compilers using syntactic skeletons derived from their own regression test-suites and find 217 confirmed bugs in GCC/Clang compiler. Jattack[86] was primarily developed to complement manually-written tests. Developers can embed their knowledge into program generation by specifying holes for exploration, enabling better testing of JIT compilers that require complex structures and execution to reveal bugs.

OBsmith uses sketching to catch the domain knowledge. Program sketching [5, 26, 29, 63, 64, 66–69], pioneered by the Sketch system [66], offered an exciting new advance in scaling program synthesis, where a partial implementation is given and the goal is to complete it [9, 17, 18, 21, 24, 35, 36, 47, 53, 63]. EdSketch [27] and EdSynth [81] defined an optimized backtracking search for completing Java sketches where test executions guided search pruning. The Sketch system provided

Java-like Sketch language for writing partial programs, and deployed SAT and inductive synthesis in a counterexample-guided loop to complete them. JSketch enabled sketching Java programs [29] by translating Java to Sketch. PSketch focused on concurrent data structures and enabled sketching them [68].

7.3 Large Language Models

LLMs have recently demonstrated strong capabilities in diverse code-related tasks, and they have enabled the automation of many software development and verification themes, including writing code [6, 77, 91, 92], clarifying requirements [51], software maintenance [14, 32, 33, 45, 52, 79], software testing [15, 31, 41, 60], debugging [34, 71], constructing proofs of theorems in automated provers [19], and human-centric studies [28, 76].

Trained on extensive corpora of code and textual data, these models develop an implicit understanding of programming language syntax, semantics, and code structures[30, 40, 59]. Consequently, LLMs exhibit notable capabilities in tasks requiring both natural language comprehension and code manipulation[31]. While models such as StarCoder[42] and Gemini demonstrate proficiency in code generation, they remain susceptible to hallucination, wherein they generate plausible but incorrect or nonsensical outputs[22]. Moreover, LLMs exhibit limitations in tasks demanding precise logical and mathematical reasoning[83, 90].

Although LLMs have shown to be effective in the above tasks, there is still a lack of research demonstrating their ability to generate sketches. To the best of our knowledge, OBsmith presents the first study of LLMs in sketching problems for obfuscator testing. We show that LLMs are capable of generating high-quality sketches thanks to a large training corpus containing domain knowledge.

8 Conclusion

In this paper, we present OBsmith, a framework that enables LLM-powered JavaScript obfuscator testing. Using OBsmith, obfuscator developers can use LLMs to generate representative JavaScript sketches which are suitable for testing obfuscators. OBsmith fills sketches and enhances them to enable differential testing. OBsmith also expose program generator as a script to allow developers write their sketches in the same language as the obfuscators they are testing (JavaScript), enabling them to leverage their domain knowledge to set up a code structure likely to lead to obfuscation problem while leaving stakeholders representing expressions they want explored. Using 600 sketches generated by 6 LLMs, OBsmith generates 3,000 programs and found 11 bugs in 2 popular JavaScript obfuscators used by malware developers. These bugs cover obfuscator crash, obfuscated code crash, functionality changes, etc. OBsmith combines the power of LLM to synthesize representative sketches and perform random differential testing to detect critical vulnerabilities.

9 Data-Availability Statement

We will provide the artifact with all prompts, sketches, programs, and bug report. We will also provide a ready to use artifact to run our sketch filling algorithm, enhance program algorithm, and testing scripts.

References

[1] 2014. Babel: A tool that helps you write code in the latest version of JavaScript. https://github.com/babel/babel.
[2] 2020. jsfunfuzz. https://github.com/MozillaSecurity/funfuzz.
[3] 2022. js-confuser. https://github.com/MichaelXF/JS-Confuser.

[4] 2024. JavaScript-obfuscator: A Powerful Obfuscator for JavaScript and Node.js. https://github.com/javascript-obfuscator/javascript-obfuscator.

[5] Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *FMCAD*.

[6] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. 2024. CodePlan: Repository-Level Coding using LLMs and Planning. FSE (2024).

[7] Salman A. Baset, Shih-Wei Li, Philippe Suter, and Omer Tripp. 2017. Identifying Android library dependencies in the presence of code obfuscation and minimization *(ICSE-C '17)*.

[8] Gregory Blanc, Daisuke Miyamoto, Mitsuaki Akiyama, and Youki Kadobayashi. 2012. Characterizing Obfuscated JavaScript Using Abstract Syntax Trees: Experimenting with Malicious Scripts. In *WAINA*.

[9] Rastislav Bodík and Barbara Jobstmann. 2013. Algorithmic program synthesis: Introduction. *STTT* (2013).

[10] Douglas Brewer, Kang Li, Laksmish Ramaswamy, and Calton Pu. 2010. A link obfuscation service to detect webbots. In *SCC*.

[11] Kenneth Brezinski and Ken Ferens. 2023. Metamorphic malware and obfuscation: a survey of techniques, variants, and generation kits. *Security and Communication Networks* 2023, 1 (2023), 8227751.

[12] Gerardo Canfora, Andrea Di Sorbo, Francesco Mercaldo, and Corrado Aaron Visaggio. 2015. Obfuscation techniques against signature-based detection: a case study. In *2015 Mobile systems technologies workshop (MST)*. IEEE, 21–26.

[13] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2011. ZOZZLE: fast and precise in-browser JavaScript malware detection *(SEC'11)*.

[14] Malinda Dilhara, Abhiram Bellur, Timofey Bryksin, and Danny Dig. 2024. Unprecedented Code Change Automation: The Fusion of LLMs and Transformation by Example. FSE (2024).

[15] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. TOGA: A neural method for test oracle generation. In *ICSE*. 2130–2141.

[16] Tony Doyle. 2018. Privacy, obfuscation, and propertization. *IFLA Journal* (2018), 229 – 239.

[17] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based Synthesis for Complex APIs. In *POPL*.

[18] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-output Examples. In *PLDI*.

[19] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-Proof Generation and Repair with Large Language Models. In *ESEC/FSE*.

[20] Daniel Fraunholz and Hans D. Schotten. 2018. Defending Web Servers with Feints, Distraction and Obfuscation. In *ICNC*.

[21] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. 2014. CodeHint: Dynamic and Interactive Synthesis of Code Snippets. In *ICSE*.

[22] Andrew Gambardella, Yusuke Iwasawa, and Yutaka Matsuo. 2024. Language Models Do Hard Arithmetic Tasks Easily and Hardly Do Easy Arithmetic Tasks. In *ACL*.

[23] Samuel Groß. 2018. Fuzzil: Coverage guided fuzzing for javascript engines. *Department of Informatics, Karlsruhe Institute of Technology* (2018).

[24] Tihomir Gvero, Viktor Kuncak, and Ruzica Piskac. 2011. Interactive Synthesis of Code Snippets. In *CAV*.

[25] Mahmoud Hammad, Joshua Garcia, and Sam Malek. 2018. A large-scale empirical study on the effects of code obfuscations on Android apps and anti-malware products *(ICSE)*.

[26] Yang Hong, Shan Jiang, Yulei Fu, and Sarfraz Khurshid. 2025. On the Effectiveness of Large Language Models in Writing Alloy Formulas. *arXiv preprint arXiv:2502.15441* (2025).

[27] Jinru Hua and Sarfraz Khurshid. 2017. EdSketch: Execution-Driven Sketching for Java. In *SPIN*.

[28] Mia Mohammad Imran, Preetha Chatterjee, and Kostadin Damevski. 2024. Uncovering the Causes of Emotions in Software Developer Communication Using Zero-shot LLMs. In *ICSE*.

[29] Jinseong Jeon, Xiaokang Qiu, Jeffrey S. Foster, and Armando Solar-Lezama. 2015. JSketch: Sketching for Java. In *FSE*.

[30] Shan Jiang, Pranoy Kovuri, David Tao, and Zhixun Tan. 2025. CASCADE: LLM-Powered JavaScript Deobfuscator at Google. *arXiv preprint arXiv:2507.17691* (2025).

[31] Shan Jiang, Chenguang Zhu, and Sarfraz Khurshid. 2024. Generating executable oracles to check conformance of client code to requirements of JDK Javadocs using LLMs. *arXiv preprint arXiv:2411.01789* (2024).

[32] Zhihan Jiang, Jinyang Liu, Zhuangbin Chen, Yichen Li, Junjie Huang, Yintong Huo, Pinjia He, Jiazhen Gu, and Michael R. Lyu. 2024. LILAC: Log Parsing using LLMs with Adaptive Parsing Cache. FSE (2024).

[33] Xin Jin and Zhiqiang Lin. 2024. SimLLM: Calculating Semantic Similarity in Code Summaries using a Large Language Model-Based Approach. FSE (2024).

[34] Sungmin Kang, Gabin An, and Shin Yoo. 2024. A Quantitative and Qualitative Evaluation of LLM-Based Explainable Fault Localization. FSE (2024).

[35] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis Modulo Recursive Functions. In *OOPSLA*.
[36] Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter. 2010. Complete Functional Synthesis. In *PLDI*.
[37] Bo Li, Phani Vadrevu, Kyu Hyung Lee, and Roberto Perdisci. 2018. JSgraph: Enabling Reconstruction of Web Attacks via Efficient Tracking of Live In-Browser JavaScript Executions. In *NDSS*.
[38] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. 2021. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *USENIX Security (SEC '21)*.
[39] Zhihao Li, Chaozheng Wang, Zongjie Li, Xinyong Peng, Zelin Su, Qun Xia, Haochuan Lu, Ting Xiong, Man Ho Lam, Shuzheng Gao, et al. 2025. JSProtect: A Scalable Obfuscation Framework for Mini-Games in WeChat. *arXiv preprint arXiv:2509.24498* (2025).
[40] Kaibo Liu, Yiyang Liu, Zhenpeng Chen, Jie M Zhang, Yudong Han, Yun Ma, Ge Li, and Gang Huang. 2024. LLM-Powered Test Case Generation for Detecting Tricky Bugs. *arXiv preprint arXiv:2404.10304* (2024).
[41] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2024. Make LLM a Testing Expert: Bringing Human-like Interaction to Mobile GUI Testing via Functionality-aware Decisions. In *ICSE*.
[42] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173* (2024).
[43] Yifei Lu, Weidong Hou, Minxue Pan, Xuandong Li, and Zhendong Su. 2024. Understanding and finding Java decompiler bugs. OOPSLA (2024).
[44] Benjamin Lynn, Manoj Prabhakaran, and Amit Sahai. 2004. Positive results and techniques for obfuscation. In *EUROCRYPT*. Springer, 20–39.
[45] Zeyang Ma, An Ran Chen, Dong Jae Kim, Tse-Hsun Chen, and Shaowei Wang. 2024. LLMParser: An Exploratory Study on Using Large Language Models for Log Parsing. In *ICSE*.
[46] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. 2015. Stealth attacks: An extended insight into the obfuscation effects on Android malware. *Computers & Security* (2015).
[47] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid mining: helping to navigate the API jungle. In *PLDI*. 48–61.
[48] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* (1998).
[49] Chuize Meng, Shan Jiang, Mengning Wu, Xuan Xiao, Dan Tao, and Ruipeng Gao. 2022. BatMapper-Plus: Smartphone-Based Multi-level Indoor Floor Plan Construction via Acoustic Ranging and Inertial Sensing.
[50] Chuize Meng, Shan Jiang, Mengning Wu, Xuan Xiao, Dan Tao, and Ruipeng Gao. 2023. Smartphone-Based Indoor Floor Plan Construction via Acoustic Ranging and Inertial Tracking. *Machines* (2023).
[51] Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binquan Zhang, ChenXue Wang, Shichao Liu, and Qing Wang. 2024. ClarifyGPT: A Framework for Enhancing LLM-Based Code Generation via Requirements Clarification. FSE (2024).
[52] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to Help With Code Understanding. In *ICSE*. Article 97, 13 pages.
[53] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed Program Synthesis. In *PLDI*.
[54] Nikolaos Pantelaios and Alexandros Kapravelos. 2024. FV8: A Forced Execution JavaScript Engine for Detecting Evasive Techniques. *arXiv preprint arXiv:2405.13175* (2024).
[55] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *IEEE Symposium on Security and Privacy (Oakland)*.
[56] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016. Learning programs from noisy data. (2016), 761–774.
[57] Kunlun Ren, Weizhong Qiang, Yueming Wu, Yi Zhou, Deqing Zou, and Hai Jin. 2023. JSRevealer: A Robust Malicious JavaScript Detector against Obfuscation. In *DSN*.
[58] Xiaotong Ren, Shuli Zhu, Chuize Meng, Shan Jiang, Xuan Xiao, Dan Tao, and Ruipeng Gao. 2022. PeTrack: Smartphone-based Pedestrian Tracking in Underground Parking Lot. In *MSN*.
[59] Jonan Richards, Mairieli Wessel, and Hanna Schraffenberger. 2024. What You Need is What You Get: Theory of Mind for LLM-Generated Code Explanations. (2024).
[60] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. 2024. Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM. FSE (2024).
[61] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. 2022. Loki: Hardening Code Obfuscation Against Automated Attacks. In *SEC*.

[62] Mayank Sharma, Pingshi Yu, and Alastair F Donaldson. 2023. Rustsmith: Random differential compiler testing for rust. In *ISSTA*.
[63] Rishabh Singh and Sumit Gulwani. 2015. Predicting a Correct Program in Programming by Example. In *CAV*.
[64] Rishabh Singh and Armando Solar-Lezama. 2011. Synthesizing Data Structure Manipulations from Storyboards. In *FSE*.
[65] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. 2019. Anything to hide? studying minified and obfuscated code in the web. In *WWW*.
[66] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph. D. Dissertation. University of California, Berkeley.
[67] Armando Solar-Lezama, Gilad Arnold, Liviu Tancau, Rastislav Bodik, Vijay Saraswat, and Sanjit Seshia. 2007. Sketching Stencils. *PLDI* (2007).
[68] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. 2008. Sketching Concurrent Data Structures. In *PLDI*.
[69] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *ASPLOS*.
[70] Ruoxi Sun, Minhui Xue, Gareth Tyson, Tian Dong, Shaofeng Li, Shuo Wang, Haojin Zhu, Seyit Camtepe, and Surya Nepal. 2023. Mate! Are You Really Aware? An Explainability-Guided Testing Framework for Robustness of Malware Detectors *(ESEC/FSE 2023)*.
[71] Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani. 2024. CORE: Resolving Code Quality Issues using LLMs. FSE (2024).
[72] Haoyu Wang, Junjie Chen, Chuyue Xie, Shuang Liu, Zan Wang, Qingchao Shen, and Yingquan Zhao. 2023. Mlirsmith: Random program generation for fuzzing mlir compiler infrastructure. In *ASE*.
[73] 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)*. IEEE, 724–735.
[74] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. 2023. FuzzJIT: Oracle-enhanced fuzzing for JavaScript engine JIT compiler. In *USENIX Security (SEC '23)*. USENIX Association.
[75] Kaiyuan Wang, Allison Sullivan, Darko Marinov, and Sarfraz Khurshid. 2018. Solver-based Sketching of Alloy Models using Test Valuations. In *ABZ*.
[76] Wei Wang, Huilong Ning, Gaowei Zhang, Libo Liu, and Yi Wang. 2024. Rocks Coding, Not Development: A Human-Centric, Experimental Evaluation of LLM-Supported SE Tasks. FSE (2024).
[77] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2025. Agentless: Demystifying LLM-based Software Engineering Agents. In *ESEC/FSE*. 24 pages.
[78] Zifan Xie, Ming Wen, Haoxiang Jia, Xiaochen Guo, Xiaotong Huang, Deqing Zou, and Hai Jin. 2023. Precise and Efficient Patch Presence Test for Android Applications against Code Obfuscation *(ISSTA 2023)*.
[79] Junjielong Xu, Ziang Cui, Yuan Zhao, Xu Zhang, Shilin He, Pinjia He, Liqun Li, Yu Kang, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. 2024. UniLog: Automatic Logging via LLM and In-Context Learning. In *ICSE*.
[80] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI*.
[81] Zijiang Yang, Jinru Hua, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Test Execution Driven Synthesis of API Sequences With Conditionals and Loops. In *ICST*.
[82] Pingshi Yu, Nicolas Wu, and Alastair F Donaldson. 2025. Ratte: Fuzzing for Miscompilations in Multi-Level Compilers Using Composable Semantics. In *ASPLOS*.
[83] Zheng Yuan, Hongyi Yuan, Chuanqi Tan, Wei Wang, and Songfang Huang. 2023. How well do large language models perform in arithmetic tasks? *arXiv preprint arXiv:2304.02015* (2023).
[84] Zhiqiang Zang, Aditya Thimmaiah, and Milos Gligoric. 2024. JOG: Java JIT peephole optimizations and tests from patterns. In *ICSE*.
[85] 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.
[86] Zhiqiang Zang, Fu-Yao Yu, Nathan Wiatrek, Milos Gligoric, and August Shi. 2023. JAttack: Java JIT Testing Using Template Programs *(ICSE '23)*.
[87] Jiexin Zhang, Alastair R. Beresford, and Stephan A. Kollmann. 2019. LibID: reliable identification of obfuscated third-party Android libraries *(ISSTA 2019)*.
[88] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *PLDI*.
[89] Xiaolu Zhang, Frank Breitinger, Engelbert Luechinger, and Stephen O'Shaughnessy. 2021. Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. *Forensic Science International: Digital Investigation* (2021).
[90] Yilun Zhao, Yitao Long, Hongjun Liu, Ryo Kamoi, Linyong Nan, Lyuhao Chen, Yixin Liu, Xiangru Tang, Rui Zhang, and Arman Cohan. 2024. Docmath-eval: Evaluating math reasoning capabilities of llms in understanding financial

documents. In *ACL*.
[91] Hua Zhong, Shan Jiang, and Sarfraz Khurshid. 2025. An approach for API synthesis using large language models. *arXiv preprint arXiv:2502.15246* (2025).
[92] Hua Zhong, Shan Jiang, and Sarfraz Khurshid. 2025. APRIL: API Synthesis with Automatic Prompt Optimization and Reinforcement Learning. *arXiv preprint arXiv:2509.25196* (2025).

10 Appendix

```
<context>
    You are a software testing engineer. Now your task is to generate some template
    JavaScript programs, and these programs will be used to test JavaScript obfuscators. In
    template, you should use holes to represent a holder, these holes can be numberLiteral,
    or booleanLiteral. And these holes can also be numberReference, or booleanReference. And
    you can use binary expression including arithmetic operation(e.g. +, -, *, /,  ...),
    relation operation(>, <, ==, ===, >=, <=, ...) and logic operation (AND, OR, NOT, ...)
    in templates, the operands can be any type of literals, references, or holes. I will use
    templates you generate to do differential testing. Specifically, I will fill holes to
    get programs and run the filled programs with JavaScript obfuscators and minifiers.
    Finally I will compare if the transformed JavaScript program have the same output of
    original filled program.
</context>

<example>
    <template>
        let s1 = numberLiteral;
        let s2 = numberLiteral;
        let b1 = booleanLiteral;
        let arr = [s1++, s2, numberLiteral, numberLiteral, booleanLiteral]

        for (int i = 0; i < arr.length; i++) {
          if (logic(relation(numberReference, numberReference, <=),
                    relation(numberReference, numberReference, <=),
                             &&, ||)){
            arr[i] = arithmetic(numberReference, numberReference, +, *);
          }
          if (booleanReference) {
            booleanReference = booleanLiteral;
            booleanReference = booleanReference;
          }
        }
    </template>
    <filled program>
        let s1 = 1098733;
        let s2 = 38792312;
        let b1 = true;
        let arr = [s1++, s2, 37289, 776, true]

        for (int i = 0; i < arr.length; i++) {
          if (arr[1] <= s2 || s2 <= arr[0]){
            arr[i] = arr[0] + s1;
          }
          if (arr[4]){
            b1 = false;
            b1 = arr[4];
          }
        }
    </filled program>
</examples>

<format>
    You should output sketches in a markdown block without any explaination. e.g.:
    ```javascript
        <sketch 1>
        ...
        <sketch 10>
    ```
</format>

<instruction>
    Now please generate <number> different templates for me. You should use templates to
    cover as many representative javascript programs as possible, including different
    control flows, common functions, etc. Make sure your output follows format instruction.
</instruction>
```

Fig. 5. OBsmith prompt for LLM-based sketch generation