

# CASCADE: LLM-powered JavaScript Deobfuscator at Google

Shan Jiang\*

The University of Texas at Austin  
Austin, Texas, USA  
shanjiang@utexas.edu

David Tao

Google Inc  
Sunnyvale, California, USA  
dtao@google.com

Pranoy Kovuri\*

Google Inc  
Sunnyvale, California, USA  
kovuripranoy@gmail.com

Zhixun Tan

Google Inc  
Sunnyvale, California, USA  
tzx@google.com

## Abstract

Software obfuscation, particularly prevalent in JavaScript, hinders code comprehension and analysis, posing significant challenges to software testing, static analysis, and malware detection. This paper introduces CASCADE, a novel hybrid approach that integrates the advanced coding capabilities of Gemini with the deterministic transformation capabilities of a compiler Intermediate Representation (IR), specifically JavaScript IR (JSIR). By employing Gemini to identify critical prelude functions—the foundational components underlying the most prevalent obfuscation techniques—and leveraging JSIR for subsequent code transformations, CASCADE effectively recovers semantic elements like original strings and API names, and reveals original program behaviors. This method overcomes limitations of existing static and dynamic deobfuscation techniques, eliminating hundreds to thousands of hard-coded rules while achieving reliability and flexibility. CASCADE is already deployed in Google’s production environment, demonstrating substantial improvements in JavaScript deobfuscation efficiency and reducing reverse engineering efforts.

## CCS Concepts

• **Software and its engineering** → **Compilers**; • **Security and privacy** → **Software reverse engineering**; • **Computing methodologies** → **Artificial intelligence**.

## Keywords

JavaScript Deobfuscation, Large Language Models, Compiler

### ACM Reference Format:

Shan Jiang, Pranoy Kovuri, David Tao, and Zhixun Tan. 2026. CASCADE: LLM-powered JavaScript Deobfuscator at Google. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-SEIP ’26)*, April 12–18, 2026, Rio de Janeiro, Brazil. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3786583.3786873>

## 1 Introduction

\*Work done while at Google.



This work is licensed under a Creative Commons Attribution 4.0 International License. *ICSE-SEIP ’26, Rio de Janeiro, Brazil*

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2426-8/26/04

<https://doi.org/10.1145/3786583.3786873>

JavaScript is the dominant programming language for web development, powering client-side interactions across billions of web pages, mobile applications, and browser extensions. However, its widespread adoption has led to increased use of code obfuscation techniques that deliberately transform readable code into complex, difficult-to-understand variants. This poses significant challenges to software testing, complicates analysis, and hinders malware detection. Deobfuscation, which transforms the code to restore readability and reveal the original intention while maintaining code semantic equivalence, is inherently difficult. It requires robust handling of various complex obfuscation techniques—such as dynamic code generation (e.g., using the eval function to execute code from strings), control flow flattening, and string encoding. Deobfuscation involves not only syntactic simplification but also deeper semantic restructuring to reverse transformations like identifier mangling, opaque predicate removal, and constant unfolding. Aggressive code transformations might significantly enhance readability but inadvertently compromise semantic equivalence; conservative approaches may produce code that, while correct, remains largely inscrutable.

Based on empirical observations, Obfuscator.IO [3] is the most widely used JavaScript obfuscator by malware developers, with js-confuser [6] is the second most common. Obfuscator.IO’s level of obscurity, as demonstrated in a concrete example shown in Fig 1, significantly impedes malware detection workflows. Amongst its various obfuscation techniques, the obfuscation of strings and API names (method and property names) creates the greatest barriers to code analysis. Recovering original string literals and API names would markedly reduce the manual effort required in Google’s malware detection processes. Since Obfuscator.IO performs API obfuscation by transforming direct API calls (e.g., chrome.cookies) as string-indexed lookups (chrome["cookies"]), solving string obfuscation inherently resolves the API obfuscation. Consequently, we target Obfuscator.IO’s string obfuscation in this paper.

To address string obfuscation of Obfuscator.IO, we propose CASCADE (Combined Analysis of Scripts with a Context-Aware Deobfuscation Engine), a hybrid JavaScript deobfuscator that combines large language models (LLMs) and a compiler intermediate representation (IR). First, it uses Gemini [17] to detect and extract key code patterns, termed *prelude functions*, generated by Obfuscator.IO for manipulating obfuscated strings. Then, it runs a customized constant propagation and inline pass built on JSIR [8], a next-generation compiler framework, and treats prelude functions

```

function hi() {
  console.log('Hello World!');
}
hi();

function _0x432d() {
  var _0x1398fd = [
    '754705BaCmnb', '710BeTfxi', '6078JFPbmg', '391457WTcWXy', '6916iSWoZ0',
    '1533357iISYWF', '2400oHfZQf', '24eeZIfI', 'log', '149YcoFBV',
    '1367628nfkDqA', '1243948epsCBk', 'Hello\x20World!'
  ];
  _0x432d = function () {
    return _0x1398fd;
  };
  return _0x432d();
}
(function (_0x38057e, _0xee6281) {
  var _0x18e9b0 = _0x4c0c;
  var _0x1ab84e = _0x38057e();
  while (!![]) {
    try {
      var _0xc7d52d = parseInt(_0x18e9b0(0x1b7)) / 0x1 * (-parseInt(_0x18e9b0(0x1bf)) / 0x2) + -parseInt(_0x18e9b0(0x1b8)) / 0x3 + -
      parseInt(_0x18e9b0(0x1b9)) / 0x4 + -parseInt(_0x18e9b0(0x1c1)) / 0x5 * (parseInt(_0x18e9b0(0x1bd)) / 0x6) + parseInt(_0x18e9b0(0x1bb))
      / 0x7 * (-parseInt(_0x18e9b0(0x1b5)) / 0x8) + -parseInt(_0x18e9b0(0x1c0)) / 0x9 + -parseInt(_0x18e9b0(0x1bc)) / 0xa * (-parseInt(
      _0x18e9b0(0x1be)) / 0xb);
      if (_0xc7d52d === _0xee6281) {
        break;
      } else {
        _0x1ab84e['push'](_0x1ab84e['shift']());
      }
    } catch (_0x5a174b) {
      _0x1ab84e['push'](_0x1ab84e['shift']());
    }
  }
})(_0x432d, 0x40942);
function _0x4c0c(_0x32b956, _0x514a26) {
  var _0x432d26 = _0x432d();
  _0x4c0c = function (_0x4c0c62, _0x284b04) {
    _0x4c0c62 = _0x4c0c62 - 0x1b5;
    var _0x85349e = _0x432d26[_0x4c0c62];
    return _0x85349e;
  };
  return _0x4c0c(_0x32b956, _0x514a26);
}
function hi() {
  var _0x964834 = _0x4c0c;
  console[_0x964834(0x1b6)](_0x964834(0x1ba));
}
hi();

```

**Figure 1: Hello World obfuscated by Obfuscator.IO with the default (the lowest level) configuration**

as pure (i.e. idempotent and side-effect-free) functions and dynamically execute them in a sandboxed JavaScript environment. The use of LLMs for code pattern detection eliminates reliance on manually engineered heuristics; the compiler IR provides a robust structure for applying code transformations, ensuring functional integrity through semantics-preserving operations while systematically improving code readability.

Recent work confirms that LLMs possess strong code understanding skills [25, 31, 37], making them promising tools for identifying code patterns. LLMs also align user prompts with their learned internal representations [27] and show emergent competence on tasks not seen during training [46]. Together, these properties position LLMs to automatically pinpoint prelude functions in long, obfuscated codebases. However, LLMs still falter on tasks that demand exact logical or mathematical reasoning [50, 53] limiting their effectiveness in fully correct end-to-end deobfuscation. Even small arithmetic mistakes—e.g., flipping an if condition from true to false—can radically change program behavior. Evaluating pure LLM deobfuscation is also difficult because proving functional equivalence between the obfuscated and recovered programs is non-trivial.

The hybrid approach of CASCADE leverages the best of both worlds: Gemini detects prelude functions with a 99.56% average success rate on a synthetic dataset of 12K obfuscated files; the deterministic transformations of JSIR then restores 945 string literals per sample in an average of two seconds, demonstrating practical throughput.

This paper offers three primary contributions:

- **Novelty** — CASCADE is the first framework to pair an LLM with compiler-level IR transformations, merging probabilistic code understanding with deterministic rewrites for robust deobfuscation of JavaScript code.
- **Industrial deployment** — CASCADE is deployed in production at Google and eliminates the need for hard-coded pattern matching rules (100+ lines per obfuscation technique) in WebCrack and Deobfuscator.IO.
- **Robustness** - CASCADE hybrid architecture eliminates hard-coded pattern matching which can be easily broken with just a slight change after obfuscation, and is robust to regression changes of Obfuscator.IO.
- **Scalability** — We open-source our prompt templates and the full JSIR infrastructure to facilitate community adoption

and reproducibility. The repository is publicly available at <https://github.com/google/jsir>.

## 2 Background and Related Work

### 2.1 Software Obfuscation.

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 [14, 52]. Although obfuscation can legitimately safeguard proprietary algorithms and sensitive resources (e.g., IP addresses) [19, 35], it is also exploited by attackers to mask malicious logic—especially in web and mobile scripts [15, 36, 47].

Obfuscated code severely diminishes the effectiveness of analysis tools, creating a major obstacle for software testing and static analysis methods [12, 49, 51]. Moreover, obfuscation hampers malware detection by concealing malicious behavior from static analyzers, security filters, machine-learning detectors, and manual reviewers [30, 40, 44]. JavaScript, the dominant client-side language, is particularly vulnerable because its source code is delivered directly to the browser [18, 21], making it an attractive target for both defensive and malicious obfuscation.

### 2.2 JavaScript Deobfuscation

The prevalence of malicious JavaScript code underscores an urgent need for effective deobfuscation techniques [16, 26, 43]. JavaScript deobfuscators aim to restore code readability for analysis while preserving semantic correctness. Various deobfuscation approaches have been explored in recent years.

Machine learning-based approaches, exemplified by DEGUARD [13] for ProGuard-obfuscated Android code, have demonstrated promising results. However, the inherent stochasticity of current ML models prevents them from guaranteeing semantic equivalence between original and deobfuscated code.

Dynamic analysis can enhance code readability to facilitate manual inspection of obfuscated scripts [24]. Lu et al. [34] proposed a dynamic analysis technique incorporating program slicing. However, dynamic methods generally impose specific runtime environment requirements, incur substantial performance overhead, and raise security concerns.

Static analysis presents an alternative approach. JSDES [11] introduced function-centric deobfuscation but struggles with obfuscations implemented purely through basic operations. This limitation renders it less effective against Obfuscator.IO, which commonly interleaves such operations with complex function calls. Many static techniques operate at the AST level [5, 7, 10, 20]. TransAST [42] uses static analysis and machine translation to deobfuscate JavaScript but faces difficulties with obfuscation techniques that are dynamically generated or manifested at runtime.

Pattern-matching approaches, such as Safe-Deobs [24], rely on predefined patterns derived from real-world malware. Webcrack [7] is a rule-based JavaScript deobfuscator that transforms code at the AST level. Webcrack relies on hard-coded AST rules, so even slight modifications (e.g., changing `while (![])` to `while (!false)`) can

prevent successful deobfuscation. AST-based methods are limited when encountering heavily obfuscated code or novel obfuscation patterns not included in their predefined rule sets. The key limitation of AST-based tools is their lack of semantic understanding capabilities compared to compiler-based methods. These tools are prone to altering code behavior or introducing errors during the deobfuscation process.

### 2.3 LLM for SE.

Large Language Models (LLMs) have recently demonstrated strong capabilities in diverse code-related tasks, including program synthesis, code refactoring, and automated test generation [27, 31, 54, 55]. Trained on extensive corpora of code and textual data, these models develop an implicit understanding of programming language syntax, semantics, and code structures [28, 32, 45]. Consequently, LLMs exhibit notable capabilities in tasks requiring both natural language comprehension and code manipulation [27]. While models such as StarCoder [33] 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 [50, 53]. This constrains their efficacy in achieving correct end-to-end deobfuscation, where precise computation is crucial, as even minor inaccuracies can drastically alter program behavior. To address these non-deterministic limitations, hybrid approaches integrating LLMs with Abstract Syntax Trees (ASTs) or Intermediate Representations (IRs) have demonstrated promise in various software engineering applications [39].

## 3 String Obfuscation And CASCADE Approach

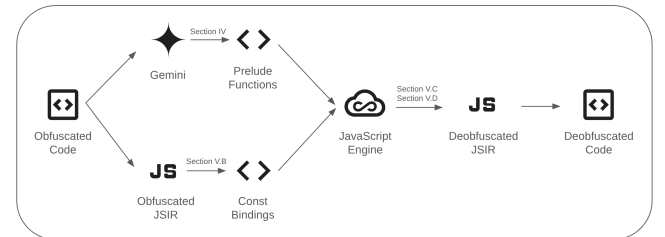


Figure 2: CASCADE Deobfuscator

Obfuscator.IO represents a widely adopted obfuscation tool among malicious actors [9]. The tool employs a multi-stage obfuscation pipeline incorporating string obfuscation, control flow flattening, and complex arithmetic operations, etc.. String obfuscation emerges as the most critical technique for two key reasons: (1) recovering string literals and API names yields the greatest readability improvements, and (2) string obfuscation occurs as the final pipeline stage, layering atop other obfuscation methods and establishing string deobfuscation as a prerequisite for subsequent analysis. **Consequently, CASCADE prioritizes string deobfuscation when processing Obfuscator.IO-generated code.**

The Obfuscator.IO tool implements string obfuscation through two code transformations. First, it creates a **string fetching function** that retrieves and optionally decodes elements from a **global**

**string table.** Second, it replaces each original string literal with a complex **string recovery expression** that calls the fetching function through multiple abstraction layers including wrapper functions, object property lookups, variable aliases, and arithmetic operations.

Figure 3 is a simplified version of Figure 1, and illustrates string obfuscation in Obfuscator.IO. In this example, the strings "log" and "Hello World!" are replaced by calls to a `getString()` function, which retrieves the corresponding entries from a global string array using offset indices.

```
function hi() {
  console.log('Hello World!');
}
hi();

var globalStringTable = [
  ..., /*index: 8*/'log',
  ..., /*index: 12*/'Hello World!'
];
function getStringArray() {
  return globalStringTable;
}

function getString(index) {
  return getStringArray()[index - 437];
}

(function () {
  var stringArray = getStringArray();
  while (true) {
    if (conditionOnPermutationOf(stringArray)) {
      stringArray.push(stringArray.shift());
    }
  }
})();

function hi() {
  console[getString(438)][getString(442)];
}
hi();
```

**Figure 3: Simplified Illustration of String Obfuscation**

The global string table undergoes rotation an unknown number of times, which makes `getStringArray()`—and by extension, `getString()`—seem non-idempotent with side effects, despite their truly idempotent and side-effect-free nature. As a result, these functions bypass compiler optimizations such as constant propagation and inlining.

However, the definition and initialization of the global string table, along with the string fetching function—which we term **prelude functions**—are generated from templates and are highly recognizable. By detecting these code segments, treating `getString()` as an idempotent and side-effect-free built-in function, and dynamically executing it upon invocation, we can enhance constant propagation and inlining, thereby successfully evaluating the string recovery expressions.

To achieve this, the CASCADE deobfuscator employs a hybrid approach that integrates static analysis, dynamic execution, and AI:

- (1) We identify prelude functions by prompting Gemini, which leverages its high-level code understanding and structured output capabilities.

- (2) We dynamically execute these detected prelude functions in a sandboxed JavaScript environment to obtain accurate results from string fetching function calls.
- (3) We apply constant propagation and inlining via JSIR—a JavaScript intermediate representation built on MLIR [29]—to evaluate arithmetic expressions and resolve indirections.

This hybrid approach has the following key benefits:

- (1) **Hybrid dynamic execution:** CASCADE overcomes the conservatism of pure static analysis, which often fails against complex obfuscations to maintain soundness. By dynamically executing code snippets that appear non-idempotent but are actually idempotent, CASCADE evaluates more expressions and recovers obfuscated strings effectively.
- (2) **AI-driven maintainability:** CASCADE eliminates the need for hundreds to thousands of lines of brittle, unreadable manual rules in prelude function detection by leveraging state-of-the-art Gemini, achieving accuracy of 99.56%. Such manual rules are easy to break with minor code changes (e.g. change `true` to `!false`, or make an alias to a variable), as shown in Figure 4. In comparison, Gemini is resilient against minor changes in code format and syntax, which evades traditional detection based on AST or regex rules.
- (3) **Advanced static analysis via JSIR:** CASCADE conducts comprehensive code analyses and transformations using the robust JSIR infrastructure, a novel high-level JavaScript intermediate representation (IR). Although ASTs preserve high fidelity to source code syntax and suit source-to-source transformations [23, 38], IRs facilitate more sophisticated semantic-level analyses and transformations [41, 48]. JSIR uniquely encodes all AST syntactical information while supporting dataflow analysis.
- (4) **Responsible use of AI:** CASCADE deliberately refrains from using LLMs to directly generate deobfuscated code, eliminating a wide range of potential hallucination errors. Its hybrid design improves explainability, observability, and evaluability, ensuring practical use in production environments.

## 4 Detecting prelude functions

Prelude functions, generated from templates, are easily recognizable to reverse engineers despite obfuscation efforts. However, manual detection becomes prohibitively time-consuming when applied to lengthy obfuscated codebases. Rule-based approaches using AST patterns [7] or regular expressions [3] prove equally problematic—they require extensive manual crafting, demand ongoing maintenance, and exhibit brittleness when confronted with minor code variations.

Recent advances in LLM code comprehension capabilities, exemplified by models such as Gemini, motivated our investigation into LLM-based prelude detection. The remainder of this section is organized as follows: Section 4.1 presents a comprehensive analysis of prelude function characteristics, Section 4.2 details our prompting methodologies, and Section 4.3 outlines optimization strategies for cost reduction and error mitigation.

```

function _0x4d08() {
  var _0x235313 = [ ... ];

  _0x4d08 = function () {
    return _0x235313;
  };
  return _0x4d08();
}

while (true) { ... }

while (!![]) { ... }

while (true) { ... }

function _0x4d08() {
  var _0x235313 = [ ... ];
  var _alias = _0x235313; // Add an alias
  _0x4d08 = function () {
    return _alias;
  };
  return _0x4d08();
}

while (!false) { ... } // true => !false

while (!!true) { ... } // !![] => !!true

for (; !false; ) { ... } // Another infinite loop

```

**Figure 4: Slight changes cause rule-based pattern match to fail, left means Webcrack deobfuscation is correct, right means Webcrack cannot deobfuscate**

#### 4.1 Prelude functions

Obfuscator.IO implements string obfuscation through three prelude functions:

- (1) A string array function that defines a global string table as an array.
- (2) A string fetching function that retrieves elements from the array using shifted indices.
- (3) A string array rotate function implemented as an IIFE (immediately invoked function expression) that rotates the global string table until a complex arithmetic expression evaluates to a target value.

The following examples illustrate these prelude functions using simplified names for clarity. In practice, Obfuscator.IO generates incomprehensible function names such as `_0x746cd9`.

**String Array Function:** Figure 6 demonstrates the `getStringArray()` function, which returns a reference to a global string table containing both original string literals and those generated during obfuscation. This function employs a self-redefining pattern: upon first invocation, it redefines itself to return a reference to the string array created during that initial call. JavaScript closure mechanisms ensure that `getStringArray()` consistently returns the same array object in subsequent invocations.

**String Fetching Function:** Figure 7 demonstrates the `getString()` function, which retrieves a single element from the global string table using an index-based lookup with a fixed offset. Similar as `getStringArray()`, the `getString()` function employs self-modification during its initial execution, rendering it non-idempotent. This behavior likely serves as an anti-analysis technique designed to prevent compiler optimization through inlining. Figure 3 illustrates the default obfuscation level implementation. At elevated obfuscation levels, the global string table contains encoded strings, and `getString()` runs custom decoding algorithms (such as Base64 or RC4 decryption), significantly increasing the function’s complexity and length.

**String Array Rotate Function:** Figure 8 demonstrates an IIFE that systematically rotates the global string table until a specific mathematical expression evaluates to a target value. The expression applies `parseInt()` to selected table strings and performs arithmetic operations, making the evaluation dependent on the table’s current permutation order. Achieving the correct permutation is essential for `getString()` to successfully retrieve the intended string value.

#### 4.2 Prompt Design

We iterated over different LLM prompt designs to detect Obfuscator.IO prelude functions. Figure 5 presents our current prompt template. We adopt a few-shot learning paradigm by incorporating descriptions and examples of the code patterns we seek to detect, as well as a concrete end-to-end example. In particular:

- (1) We separate obfuscated code by top-level statements, enclosed by HTML-style comment tags (e.g., `// <0>` and `// </0>`) of IDs. This allows us to instruct the LLM to respond in the form of statement IDs, restricting the output space, minimizing the output size, and making it easy to consume the output for downstream deobfuscation steps.
- (2) We instruct the LLM to respond in a JSON format, mapping each template type to its corresponding statement ID. This structured output simplifies downstream automatic processing.
- (3) For each target pattern, we provide a natural language description, the template source code, and several concrete examples. Since Obfuscator.IO prelude functions have a limited number of variations, a few examples are sufficient to cover all preset configurations (default, low, medium, high), as demonstrated by evaluation results in section 6.2.

This description and example-oriented prompt effectively conveys the classification criteria and desired output structure of LLM without explicit rule definition. It eliminates the need for hundreds to thousands of lines of manual rules in AST patterns [7] or regex [3], while being resilient against minor formatting or structural changes, making it more maintainable and reliable.

Our prompt design exemplifies how structured instruction engineering, combined with few-shot learning, can effectively enable modern language models to undertake sophisticated code analysis tasks, presenting substantial potential benefits for automated security analysis and research.

#### 4.3 Practical Engineering Optimization

For production deployment, we implement two engineering optimizations to minimize LLM query costs and detection errors.

**Pre-LLM filtering:** We deploy a proprietary YARA [2] rule to identify Obfuscator.IO-obfuscated JavaScript before querying Gemini. This pre-filtering eliminates the majority of the millions

```

Obfuscator.io is a well-known JavaScript obfuscator.
To obfuscate strings, it usually inserts 3 templates: StringArrayTemplate, StringArrayRotateFunctionTemplate and
StringArrayCallsWrapperTemplate.
## StringArrayTemplate
StringArrayTemplate defines a string array that includes string literals from the original unobfuscated code.
It's generated using this template:
```js
{{string_array_template}}
```

An example looks like this:
```js
{{string_array_template_example}}
```

## StringArrayCallsWrapperTemplate
StringArrayCallsWrapperTemplate fetches a string from the string array given a shifted index.
Depending on the obfuscation config, StringArrayCallsWrapperTemplate may also perform custom decoding such as base64 or rc4.
It's generated using this template:
```js
{{string_array_calls_wrapper_template}}
```

### StringArrayCallsWrapperTemplate example 1:
```js
{{string_array_calls_wrapper_template_example_1}}
```

### StringArrayCallsWrapperTemplate example 2:
...

### StringArrayCallsWrapperTemplate example 3:
...

## StringArrayRotateFunctionTemplate
StringArrayRotateFunctionTemplate is a piece of code that rotates the string array defined in StringArrayTemplate.
It's generated using this template:
```js
{{string_array_rotate_function_template}}
```

An example looks like this:
```js
{{string_array_rotate_function_template_example_1}}
```

Another example looks like this:
```js
{{string_array_rotate_function_template_example_2}}
```

Now, as an obfuscator.io expert, your job is to detect these templates from a piece of obfuscated code.
In particular, the code is split into N parts, using html tags in comments with an ID.
Please return a JSON map from template name to a list of parts belonging to that template.
Here is an example. If the input is this:

<example_java_script>
// <0>
var _0x42f9c1 = _0x1e0f;
// </0>
// <1>
String Array Rotate Function ...
// </1>
// <2>
console[_0x42f9c1(337)]('Hello, world!');
// </2>
// <3>
String Fetching Function ...
// </3>
// <4>
String Array Function ...
// </4>
</example_java_script>

Output:
{
  'StringArrayTemplate': 4,
  'StringArrayRotateFunctionTemplate': 1,
  'StringArrayCallsWrapperTemplate': 3
}

Now please investigate the following code and return the output JSON.
<investigation_java_script>
{{annotated_example}}
</investigation_java_script>
**Remember:**
* The values for StringArrayTemplate, StringArrayRotateFunctionTemplate, and StringArrayCallsWrapperTemplate can never be equal because of
  the nature of obfuscation.
* Only output the json, do not output any explanation

Output:

```

Figure 5: Prompt Template For Prelude Detection



```
function getStringArray() {
  var stringArray = [
    'info', 'ewLHr', 'prototype', ...,
    'ctor(\x22retu', 'Hello\x20Worl',
    'tZhir', '__proto__', 'table'
  ];
  getStringArray = function () {
    return stringArray;
  };
  return getStringArray();
}
```

**Figure 6: String Array Function (Variable Renamed For Readability)**

```
function getString(index, ignored) {
  var stringArray = getStringArray();
  return getString = function (index, ignored) {
    index = index -
      (-0x3*0x23b + 0x1b3 + -0x1*-0x676);
    var result = stringArray[index];
    return result;
  }, getString(index, ignored);
}
```

**Figure 7: String Fetching Function (Variable Renamed For Readability)**

```
(function (getStringArray, target) {
  var stringArray = getStringArray();
  function getStringWrapper1(a, b, c, d) {
    return getString(d - 0x1e2, a);
  }
  ...
  while (!![]) {
    try {
      var value = parseInt(getStringWrapper2(-0xbc, -0xa4, -0xc9,
        -0xab)) / (-0x22da + -0x3e*-0x4d +0x1035) + -parseInt(
        getStringWrapper1(0x38b, 0x38a, 0x357, 0x376)) ...
      if (value === target)
        break;
      else
        stringArray['push'](stringArray['shift']());
    } catch (_0x5c80ce) {
      stringArray['push'](stringArray['shift']());
    }
  }
})(getStringArray, 0x7*-0x11ff9 + -0x166d*0xc7 + 0x237a14));
```

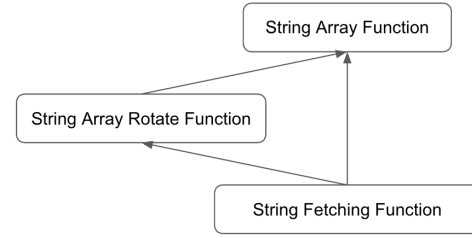
**Figure 8: String Array Rotate Function (Variable Renamed For Readability)**

of JavaScript samples processed daily, significantly reducing query costs.

**Post-LLM validation:** We set up a mechanism to validate Gemini’s result. As shown in Figure 6, the three prelude functions have a fixed dependency relationship, and do not depend on other parts of the obfuscated code. We require that the LLM detected prelude conforms to this dependency relationship, and if the verification fails, we ignore the LLM detection result.

## 5 JSIR Transformation

This section describes how we transform code and recover obfuscated strings based on the prelude detection result, through a combination of static analysis and dynamic execution using JSIR.



**Figure 9: Dependency graph of prelude functions**

We run an intra-procedural constant propagation pass augmented with (1) dynamic execution of the string fetching function, and (2) inlining of indirections introduced by Obfuscator.IO, including variable aliases, wrapper functions, and object wrappers. Table 1 illustrates the result of running the pass, assuming that dynamically executing `getString(438)` yields the string "Hello World!".

**Table 1: Illustration of JSIR transformation**

| Before  | After   |
|---|---|
| <code>var x = getString(438);</code>  | <code>var x = "Hello World!";</code>  |
| <code>// Variable alias</code><br><code>var x = getString;</code>   | <code>// Variable alias</code><br><code>var x = getString;</code>   |
| <code>...</code><br><code>var y = x(438);</code>  | <code>...</code><br><code>var y = "Hello World!";</code>  |
| <code>// Wrapper function</code><br><code>function x(a, b) {</code><br><code>  return getString(a - 1);</code><br><code>}</code>  | <code>// Wrapper function</code><br><code>function x(a, b) {</code><br><code>  return getString(a - 1);</code><br><code>}</code>  |
| <code>...</code><br><code>var y = x(439, 101);</code>   | <code>...</code><br><code>var y = "Hello World!";</code>  |
| <code>// Object wrapper</code><br><code>var o = {</code><br><code>  "k1": function (f, x) {</code><br><code>    return f(x);</code><br><code>  },</code><br><code>  "k2": function (a, b) {</code><br><code>    return a + b;</code><br><code>  }</code><br><code>};</code> | <code>// Object wrapper</code><br><code>var o = {</code><br><code>  "k1": function (f, x) {</code><br><code>    return f(x);</code><br><code>  },</code><br><code>  "k2": function (a, b) {</code><br><code>    return a + b;</code><br><code>  }</code><br><code>};</code> |
| <code>...</code><br><code>var f = getString;</code><br><code>var y = o.k1(f, o.k2(437, 1));</code>  | <code>...</code><br><code>var f = getString;</code><br><code>var y = "Hello World!";</code>   |

### 5.1 Augmenting Constant Propagation

The standard intra-procedural constant propagation is a dataflow analysis. It applies the worklist algorithm to iteratively update abstract states attached to each program point, by propagating the results of a *transfer function*, which simulates program execution in an abstract domain. In particular, we set the abstract state as a map from symbols to abstract values: {Unit, Const(some constant value), Unknown}.

Figure 10 demonstrates abstract states in action. In this example, since *a* and *b* are both known to be constants, we can calculate *d* after the first assignment; since *c* is Unknown, we have to set *d* to Unknown after the second assignment.

We augment constant propagation by extending the kinds of abstract values such that they can represent not only constant values, but also (1) references to prelude functions and (2) expressions with inline potential.

```
// State: {a#0: 100, b#0: 200, c#0: Unknown, d#0: Unknown}

d = a + b;
// State: {a#0: 100, b#0: 200, c#0: Unknown, d#0: 300}

d = a + c;
// State: {a#0: 100, b#0: 200, c#0: Unknown, d#0: Unknown}
```

Figure 10: Intra-Procedural Constant Propagation

## 5.2 Dynamic Execution of Prelude Functions

After detecting the prelude, i.e. `getStringArray`, `getString` and the string rotation code, we remove these parts from the obfuscated code, and load them in a JavaScript execution engine such as V8 [1] or QuickJS [4]. This means the JavaScript execution context now contains the rotated global string table and the definition of `getString()`. Now, utilizing the C++ API exposed by the engine, we can programmatically invoke any function defined in the context. More specifically, we can call `getString()` with any literal arguments at any time, as if `getString()` is a builtin function - this is why we call it a *prelude function*.

Then, a reference to a prelude function is considered a valid abstract value during the augmented constant propagation analysis. Figure 11 demonstrates an example, assuming `getString` is a prelude function, and the dynamic execution of `getString(438)` yields "Hello World!".

```
// State: {a#0: Unknown, f#0: Unknown}

f = getString;
// State: {a#0: Unknown, f#0: <prelude "getString">}

a = f(438);
// State: {a#0: "Hello World!", f#0: <prelude "getString">}
// State computed by dynamically executing getString(438)
```

Figure 11: Constant Propagation: Dynamic Execution Of Prelude

## 5.3 Inlining Indirections

In order to inline Obfuscator.IO indirections, we further extend the abstract value to support certain expressions with inline potential. In particular, we support the kinds of expressions listed in Figure 12.

```
expr ::= string
      | number
      | unary_op expr
      | expr bin_op expr
      | identifier
      | expr[expr]           # member expression
      | { property, ... }   # object expression
      | expr(expr, ...)     # call expression
      | (identifier, ...) => expr # function expression

property ::= (string: expr)
```

Figure 12: Supported Expressions

Then, to overcome the limit of intra-procedural analysis where the abstract state only contains symbols in the current function

under analysis, we pre-build a global lookup table for all symbols that are assigned only once with a supported expression. During the analysis, we look up symbols not only from the abstract state, but also from the global lookup table.

- (1) **Variable Alias:** Figure 13 is the example of a variable alias. Note that the alias `x` is defined outside the function `foo`, but it is still available in the global lookup table.

```
// Global lookup table:
// x: <prelude "getString">
// ...

var x = getString;
...
function foo() {
  // State: {y#1: Unknown}

  var y = x(438);
  // State: {y#1: "Hello World!"}
  //
  // State is computed by:
  // - Look up `x` in the global inline map
  // - Evaluate getString(438)
  // -      = "Hello World!" <- Dynamic execution
  ...
}
```

Figure 13: Inlining Variable Alias

- (2) **Wrapper Function:** Figure 14 is the example of a wrapper function. During the analysis, we evaluate the expression `x(439, 101)` in multiple steps, involving substitution (similar to that in lambda calculus), binary expression evaluation (standard constant folding), and dynamic execution.

```
// Global lookup table:
// x: (a#1, b#1) => getString(a#1 - 1)
// ...

function x(a, b) {
  return getString(a - 1);
}
...
function foo() {
  // State: {y#2: Unknown}

  var y = x(439, 101);
  // State: {y#2: "Hello World!"}
  //
  // State is computed by:
  // - Look up `x` in the global inline map
  // - Eval ((a#1, b#1) => getString(a#1 - 1))(439, 101)
  // -      = getString(439 - 1) <- Substitution
  // -      = getString(438) <- Binary expr evaluation
  // -      = "Hello World!" <- Dynamic execution
  ...
}
```

Figure 14: Inlining Wrapper Function

- (3) **Object Wrapper:** Figure 15 shows the example of an object wrapper, which serves as a repository of utility functions. Similar to wrapper functions, during the analysis, we evaluate the expression `o.k1(f, o.k2(437, 1))` in multiple steps, which includes fetching properties from an object expression.



```
// Global lookup table:
// o: {
//   'k1': (f#1, x#1) => f#1(x#1),
//   'k2': (a#2, b#2) => a#2 + b#2
// }

var o = {
  'k1': function (f, x) {
    return f(x);
  },
  'k2': function (a, b) {
    return a + b;
  }
};
...
function foo() {
  // {State: f#3: Unknown, y#3: Unknown}

  var f = getString;
  // {State: f#3: <getString>, y#3: Unknown}

  var y = o.k1(f, o.k2(437, 1));
  // {State: f#3: <getString>, y#3: "Hello World!"}
  //
  // State calculated by:
  // - Look up `o` in the global inline map
  // - Eval {...}.k1(<getString>, {...}.k2(437, 1))
  //   = multiple steps ...
  //   = <getString>(438)
  //   = "Hello World!"
  ...
}
```

Figure 15: Inlining Object Wrapper

## 6 Evaluation

In this section, we evaluate the CASCADE deobfuscator and address the following research questions:

**RQ1: Prelude Detection.** What is the Gemini’s accuracy of prelude function detection when processing obfuscated code?

**RQ2: JSIR Transformation.** How many literals does CASCADE recover from the obfuscated code? How long does it take to complete the deobfuscation?

**RQ3: Lessons Learned.** What are the lessons learned from adopting Gemini for automating challenging software engineering tasks like code deobfuscation?

### 6.1 Experimental Setup

We constructed a dataset by randomly selecting 3,000 JavaScript samples of diverse lengths from the ETH 150k JavaScript Dataset [49,50]. We then obfuscated each sample using four different preset configurations of Obfuscator.IO (default, low, medium, and high), which theoretically generates 12,000 obfuscated code snippets. However, Obfuscator.IO failed to process some files, yielding actually 2,937 files per configuration. Table 2 shows the size distributions of original and obfuscated samples.

We instrumented Obfuscator.IO to output prelude function locations, which serve as ground truth for evaluating Gemini prelude detection. We use Gemini 2.5 Flash without thinking (1M input tokens), for its lower latency and cost - both important for production use, and found that it already works nearly perfectly.

Table 2: File Size Distributions (KB)

|          | 50p   | 90p   | 95p   | 99p     |
|----------|-------|-------|-------|---------|
| Original | 2.2   | 19.9  | 40.4  | 186.0   |
| Default  | 5.8   | 31.0  | 58.4  | 286.3   |
| Low      | 6.9   | 26.2  | 46.6  | 207.7   |
| Medium   | 28.1  | 109.7 | 201.7 | 973.0   |
| High     | 125.8 | 340.4 | 600.7 | 2,642.0 |

Table 3: RQ1: GEMINI PRELUDE FUNCTIONS DETECTION RESULT

| Configuration | Samples | Responses (/ Samples) | Correct (/ Responses) |
|---------------|---------|-----------------------|-----------------------|
| Default       | 2935    | 2935 (100%)           | 2934 (99.97%)         |
| Low           | 2935    | 2935 (100%)           | 2935 (100%)           |
| Medium        | 2935    | 2918 (99.42%)         | 2910 (99.73%)         |
| High          | 2935    | 2867 (97.68%)         | 2825 (96.54%)         |
| All           | 11740   | 11655 (99.28%)        | 11604 (99.56%)        |

### 6.2 RQ1: Prelude detection

Table 3 evaluates Gemini’s prelude detection across various configurations, with each tested on 2,935 samples (down from 2,937 samples because 2 samples could not be annotated with IDs due to preprocessing failures caused by invalid code semantics).

The Default and Low configurations both achieved a 100% response rate - meaning Gemini provided a prelude detection result for every sample; Gemini failed to return a result for some Medium and High samples, mostly due to out-of-token-limit errors. For those samples where Gemini successfully returned a result, a near-perfect 99.56% of them matched the ground truth.

### 6.3 RQ2: JSIR transformation

Table 4: RQ2: STRING RECOVERY RESULT

| Configuration | Samples | Successes (Rate %) | Avg. Recovered Literals | Avg. Running Time (s) |
|---------------|---------|--------------------|-------------------------|-----------------------|
| Default       | 2934    | 2929 (99.83%)      | 127.71                  | 0.844                 |
| Low           | 2934    | 2929 (99.83%)      | 150.05                  | 0.893                 |
| Medium        | 2934    | 2895 (98.67%)      | 1050.14                 | 2.364                 |
| High          | 2934    | 2857 (97.38%)      | 2493.29                 | 5.164                 |
| All           | 11736   | 11610 (98.93%)     | 945.26                  | 2.298                 |

In the experiment, we set a timeout of 60 seconds. If the deobfuscator crashes or does not complete after 60 seconds, CASCADE is considered to be failed. We also count the number of literals recovered by CASCADE and record the running time. Table 4 presents a summary of our string recovery results across various obfuscation configurations. Each configuration was evaluated on 2934 samples (down from 2937 since 3 samples failed to be preprocessed). Across all 11736 samples, our approach achieved an overall high success rate of 98.93% (11610 successes) with an average of 945.26 literals recovered per file and an average running time of 2.298 seconds.

## 6.4 RQ3: Lessons Learned

In this RQ, we discuss our lessons learned from using LLMs in industrial software engineering tasks, especially JavaScript deobfuscation. LLMs’ ever-growing reasoning and code understanding capabilities suggest significant potential for LLM-driven deobfuscation. Yet, our investigation reveals that a hybrid (LLM + compiler) is still required for reliable production use. In particular, it provides the following critical improvements to an LLM-only solution:

- (1) **Reduce hallucination:** Deobfuscation of a single file relies on hundreds of accurate arithmetic calculations during operations like string retrieval, parameter calculation, and dataflow analysis. Even one subtle miscalculation, such as an off-by-one error, can fundamentally alter program semantics (e.g. by inverting the result of a conditional operation). Delegating precise analysis and calculation to a compiler tool eliminates vast categories of potential errors caused by LLM hallucination, improving correctness, which is important for production use [48].
- (2) **Provide observability and explainability:** There is limited control and explanation over why and how an LLM returns a specific response, which might even be different from its own ‘thought log’. For instance, when we tasked Gemini 2.5 Pro with deobfuscating a program that outputs “world hello”, it mistakenly responded with “hello world” even though it correctly identified “world hello” in the thought chain. This discrepancy likely stems from biases in the training data - where “hello world” is significantly more prevalent. Our hybrid approach enforces a workflow such that the result is explainable and intermediate steps are observable, improving user confidence.
- (3) **Improve verifiability and evaluability:** Due to the inherent probabilistic nature of AI, an important factor to consider for production use is the cost of verifying result correctness, which, in the case of deobfuscation, is functional equivalence between obfuscated and deobfuscated code. Since an LLM-only deobfuscator can make a wide range of errors, such verification is difficult if not impossible. Limiting the non-deterministic part to prelude detection not only makes it easier for a human to verify its correctness, but also makes it possible to define a correctness metric for evaluation.
- (4) **Reduce latency and cost:** LLM-only deobfuscation requires large amounts of reasoning, as many output tokens are the result of multiple steps of calculation. Our hybrid approach enables the use of Gemini 2.5 Flash without thinking, reducing latency and token costs. Limiting LLM use to only

Obfuscator.IO pattern detection allows us to deploy CASCADE to scan millions of JavaScript files per day.

## 7 Limitation and Future Work

While CASCADE has demonstrated success in string deobfuscation, several limitations remain. This section outlines key areas we plan to explore in future:

- (1) **Agent Integration.** Instead of a workflow of predefined steps, we will evolve CASCADE into a LLM agent. In the agent, the LLM decides by itself when to invoke various JSIR-based code transformation primitives. This could scale CASCADE to other obfuscators than Obfuscator.IO without new purpose-built logic, and defeat more circumventions.
- (2) **Deployment.** CASCADE is deployed in Google’s production environment, where it is utilized to detect malicious JavaScript on Google platforms, thereby enhancing user protection. Future work includes extending its deployment to additional platforms and releasing CASCADE as an open-source artifact to promote wider adoption within the software engineering and security community.
- (3) **Enhancing CASCADE.** Currently CASCADE focuses on string obfuscation of Obfuscator.IO, which is the most important for improving readability. But other obfuscation techniques and other obfuscators also warrant attention. Future work will leverage JSIR’s robust infrastructure to address other obfuscation strategies, e.g. control-flow flattening and dead code elimination, and support more JavaScript obfuscators.

## 8 Conclusion

CASCADE is a novel hybrid approach to JavaScript deobfuscation, offering a significant advancement in addressing complex obfuscation techniques. This approach specializes in string obfuscation of Obfuscator.IO, the most popular JavaScript obfuscator by malware writers. CASCADE employs a hybrid architecture, integrating LLMs with JSIR to achieve automated and correct deobfuscation. LLMs automate the detection of prelude functions, substantially reducing manual engineering effort, while the JSIR ensures the correctness of code transformations. Consequently, CASCADE fulfills two critical requirements for deobfuscation: it enhances code readability through string recovery, and guarantees correctness by leveraging JSIR-based compiler infrastructure. We believe that our experience of a responsible combination of LLM and compiler tools can inspire other use cases in software engineering and security communities.

## Acknowledgments

We would like to thank our colleagues across Google for the support and discussions: Alex Petit-Bianco, David Sklar, Kurt Thomas, Luca Invernizzi, Saswat Anand.

## References

- [1] 2008. V8 JavaScript engine. <https://github.com/v8/v8>.
- [2] 2013. YARA: The pattern matching swiss knife for malware researchers (and everyone else). <https://virustotal.github.io/yara>.
- [3] 2016. JavaScript-obfuscator: A Powerful Obfuscator for JavaScript and Node.js. <https://github.com/javascript-obfuscator/javascript-obfuscator>.
- [4] 2019. QuickJS JavaScript Engine. <https://github.com/bellard/quickjs>.

- [5] 2021. JavaScript Deobfuscator - General purpose JavaScript deobfuscator. <https://github.com/ben-sb/javascript-deobfuscator>.
- [6] 2021. JS-Confuser. <https://github.com/MichaelXF/js-confuser>.
- [7] 2022. webcrack - a tool for reverse engineering javascript. <https://github.com/j4k0xb/webcrack>.
- [8] 2024. JSIR - An MLIR-based JavaScript Intermediate Representation. <https://github.com/google/jsir>.
- [9] 2024. Now You See Me, Now You Don't: Using LLMs to Obfuscate Malicious JavaScript. <https://unit42.paloaltonetworks.com/using-llms-obfuscate-malicious-javascript/>.
- [10] 2024. Synchrony - A simple deobfuscator for mangled or obfuscated JavaScript files. <https://github.com/relative/synchrony>.
- [11] Moataz AbdelKhalek and Ahmed Shosha. 2017. Jsdes: An automated deobfuscation system for malicious javascript. In *ARES*.
- [12] Salman A. Baset, Shih-Wei Li, Philippe Suter, and Omer Tripp. 2017. Identifying Android library dependencies in the presence of code obfuscation and minimization. In *ICSE*.
- [13] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. 2016. Statistical Deobfuscation of Android Applications. In *CCS*.
- [14] Gregory Blanc, Daisuke Miyamoto, Mitsuki Akiyama, and Youki Kadobayashi. 2012. Characterizing Obfuscated JavaScript Using Abstract Syntax Trees: Experimenting with Malicious Scripts. In *WAINA*.
- [15] Kenneth Brezinski and Ken Ferens. 2023. Metamorphic malware and obfuscation: a survey of techniques, variants, and generation kits. *Security and Communication Networks* (2023).
- [16] Guoqiang Chen, Xin Jin, and Zhiqiang Lin. 2025. JsDeObsBench: Measuring and Benchmarking LLMs for JavaScript Deobfuscation. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*. 36–50.
- [17] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Naveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. 2025. Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities. *arXiv preprint arXiv:2507.06261* (2025).
- [18] Charlie Curtis, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2011. ZOZZLE: fast and precise in-browser JavaScript malware detection. In *SEC*.
- [19] Tony Doyle. 2018. Privacy, obfuscation, and propertization. *IFLA Journal* (2018).
- [20] Aurore Fass, Michael Backes, and Ben Stock. 2019. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In *CCS*.
- [21] Daniel Fraunholz and Hans D. Schotten. 2018. Defending Web Servers with Feints, Distraction and Obfuscation. In *ICNC*.
- [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] Quinn Hanam, Fernando S. de M. Brito, and Ali Mesbah. 2016. Discovering bug patterns in JavaScript. In *FSE*.
- [24] Adrian Herrera. 2020. Optimizing away javascript obfuscation. In *SCAM*.
- [25] 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).
- [26] Yiheng Huang, Ruisi Wang, Wen Zheng, Zhuotong Zhou, Susheng Wu, Shulin Ke, Bihuan Chen, Shan Gao, and Xin Peng. 2024. SpiderScan: Practical Detection of Malicious NPM Packages Based on Graph-Based Behavior Modeling and Matching. In *ASE*.
- [27] Shan Jiang, Chengguang 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).
- [28] Shan Jiang, Chengguang Zhu, and Sarfraz Khurshid. 2025. OBsmith: Testing JavaScript Obfuscator using LLM-powered sketching. *arXiv preprint arXiv:2510.10066* (2025).
- [29] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308
- [30] 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*.
- [31] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. Cctest: Testing and repairing code completion systems. In *ICSE*.
- [32] Kaibo Liu, Zhenpeng Chen, Yiyang Liu, Jie M Zhang, Mark Harman, Yudong Han, Yun Ma, Yihong Dong, Ge Li, and Gang Huang. 2025. LLM-powered test case generation for detecting bugs in plausible programs. *ACL* (2025).
- [33] 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).
- [34] Gen Lu and Saumya Debray. 2012. Automatic simplification of obfuscated JavaScript code: A semantics-based approach. In *SERE(SSIRI)*.
- [35] Benjamin Lynn, Manoj Prabhakaran, and Amit Sahai. 2004. Positive results and techniques for obfuscation. In *International conference on the theory and applications of cryptographic techniques*.
- [36] 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).
- [37] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. 2024. Using an LLM to help with code understanding. In *ICSE*.
- [38] Samuel Ndichu, Sangwook Kim, Seiichi Ozawa, Takeshi Misu, and Kazuo Makishima. 2019. A machine learning approach to detection of JavaScript-based attacks using AST features and paragraph vectors. *Applied Soft Computing* (2019).
- [39] Stoyan Nikolov, Daniele Codecasa, Anna Sjøvall, Maxim Tabachnyk, Satish Chandra, Siddharth Taneja, and Celal Ziftci. 2025. How is Google using AI for internal code migrations? *ICSE* (2025).
- [40] Nikolaos Pantelaios and Alexandros Kapravelos. 2024. FV8: A Forced Execution JavaScript Engine for Detecting Evasive Techniques. *arXiv preprint arXiv:2405.13175* (2024).
- [41] Jihyeok Park, Jihee Park, Seungmin An, and Sukyoung Ryu. 2020. JISET: JavaScript IR-based semantics extraction toolchain. In *ASE*.
- [42] Yan Qin, Weiping Wang, Zixian Chen, Hong Song, and Shigeng Zhang. 2023. TransAST: A machine translation-based approach for obfuscated malicious javascript detection. In *DSN*.
- [43] Kunlun Ren, Weizhong Qiang, Yueming Wu, Yi Zhou, Deqing Zou, and Hai Jin. 2023. An Empirical Study on the Effects of Obfuscation on Static Machine Learning-Based Malicious JavaScript Detectors. In *ISSTA*.
- [44] Kunlun Ren, Weizhong Qiang, Yueming Wu, Yi Zhou, Deqing Zou, and Hai Jin. 2023. JSRevealer: A Robust Malicious JavaScript Detector against Obfuscation. In *DSN*.
- [45] Jonan Richards and Mairieli Wessel. 2024. What You Need is What You Get: Theory of Mind for an LLM-Based Code Understanding Assistant.
- [46] Rylan Schaeffer, Brando Miranda, and Sanmi Koyejo. 2024. Are emergent abilities of large language models a mirage? *NeurIPS* (2024).
- [47] Moritz Schloegel, Tim Blazytko, Moritz Contag, Cornelius Aschermann, Julius Basler, Thorsten Holz, and Ali Abbasi. 2022. Loki: Hardening Code Obfuscation Against Automated Attacks. In *USENIX Security*.
- [48] Marc Szafraniec, Baptiste Roziere, Hugh James Leather, Patrick Labatut, Francois Charton, and Gabriel Synnaeve. 2023. Code Translation with Compiler Representations. In *ICLR*.
- [49] 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. In *ISSTA*.
- [50] 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).
- [51] Jiexin Zhang, Alastair R. Beresford, and Stephan A. Kollmann. 2019. LibID: reliable identification of obfuscated third-party Android libraries. In *ISSTA*.
- [52] Xiaolu Zhang, Frank Breiter, 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).
- [53] 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*.
- [54] Hua Zhong, Shan Jiang, and Sarfraz Khurshid. 2025. An approach for API synthesis using large language models. *arXiv preprint arXiv:2502.15246* (2025).
- [55] Hua Zhong, Shan Jiang, and Sarfraz Khurshid. 2025. APRIL: API Synthesis with Automatic Prompt Optimization and Reinforcement Learning. *arXiv preprint arXiv:2509.25196* (2025).