# From Obfuscated to Obvious:
# A Comprehensive JavaScript Deobfuscation Tool for Security Analysis

Dongchao Zhou[1,2], Lingyun Ying[2,*], Huajun Chai[2] and Dongbin Wang[1,*]

[1]Beijing University of Post and Telecommunication {zdc, dbwang}@bupt.edu.cn

[2]QI-ANXIN Technology Research Institute {yinglingyun, chaihuajun}@qianxin.com

*Abstract*—JavaScript's widespread adoption has made it an attractive target for malicious attackers who employ sophisticated obfuscation techniques to conceal harmful code. Current deobfuscation tools suffer from critical limitations that severely restrict their practical effectiveness. Existing tools struggle with diverse input formats, address only specific obfuscation types, and produce cryptic output that impedes human analysis.

To address these challenges, we present JSIMPLIFIER, a comprehensive deobfuscation tool using a multi-stage pipeline with preprocessing, abstract syntax tree-based static analysis, dynamic execution tracing, and Large Language Model (LLM)-enhanced identifier renaming. We also introduce multi-dimensional evaluation metrics that integrate control/data flow analysis, code simplification assessment, entropy measures and LLM-based readability assessments.

We construct and release the largest real-world obfuscated JavaScript dataset with 44,421 samples (23,212 wild malicious + 21,209 benign samples). Evaluation shows JSIMPLIFIER outperforms existing tools with 100% processing capability across 20 obfuscation techniques, 100% correctness on evaluation subsets, 88.2% code complexity reduction, and over 4-fold readability improvement validated by multiple LLMs. Our results advance benchmarks for JavaScript deobfuscation research and practical security applications.

## I. INTRODUCTION

JavaScript's dynamic nature (such as dynamic typing, runtime code modification, and code generation) and widespread adoption have made it a prime target for malicious exploitation. Attackers implement complex obfuscation techniques that evade traditional static analysis [1]–[4]. Real-world campaigns (e.g., SocGholish [5]) employ multi-layered techniques, such as string encoding, function name scrambling, and anti-analysis evasion methods [5], [6], to obfuscate JavaScript code. The obfuscated code presents significant barriers to security analysis, making it extremely difficult for analysts to understand malicious behavior and develop effective countermeasures [7].

To comprehensively understand the current deobfuscation landscape, we conducted a literature review and sample analysis to identify and categorize JavaScript obfuscation techniques. Based on the fundamental aspects of code they modify, we categorized 20 distinct obfuscation techniques across four main categories: (1) Lexical-level obfuscation (5 techniques) that transforms basic lexical elements like identifiers and literals while preserving syntactic structure; (2) Syntactic-level obfuscation (6 techniques) that modifies structural organization through advanced encoding methods and functional transformations; (3) Semantic-level obfuscation (7 techniques) that fundamentally alters implementation logic through string arrays, control flow manipulation, and dynamic code generation; and (4) Multi-layer obfuscation (2 techniques) that combines multiple strategies from different levels to create highly complex transformations.

Current JavaScript deobfuscation tools, such as JStillery [8], illuminateJS [9], and JSNice [10], face three critical limitations when handling these diverse obfuscation techniques [11]. First, they struggle with diverse input formats, crashing on malformed syntax and mixed encodings. Second, they focus on specific patterns and lack multi-layer support. Static analysis tools (e.g., JSNice [10], JSbeautifier.org [12]) fail on runtime-dependent obfuscation such as dynamic code generation and eval-based transformations, while dynamic analysis approaches (e.g., synchrony [13], JS-deobfuscator [14]) struggle with control flow manipulations and string array deobfuscation. Third, they produce cryptic identifiers that impede human analysis.

To address these limitations, we propose a comprehensive deobfuscation tool, JSIMPLIFIER, which uses a multi-stage pipeline. First, we develop robust code normalization through fault-tolerant parsing, character encoding standardization, and bundler pattern unwrapping, enabling reliable processing of diverse malformed inputs that crash existing tools. Second, our deobfuscation engine adopts a dual-strategy approach that combines static Abstract Syntax Tree (AST) analysis with controlled execution monitoring to address multi-layer obfuscation complexity. Unlike existing static-only tools that fail on runtime-dependent transformations, our enhanced static analysis handles complex control flow manipulations and

---

string array deobfuscation through comprehensive expression evaluation and intelligent scope management. Unlike existing dynamic-only approaches that struggle with static patterns, our controlled execution monitoring selectively executes obfuscated code segments within sandboxed environments and integrates results back into static analysis, enabling comprehensive support for multi-layer transformations. Finally, we apply Large Language Model (LLM)-based semantic enhancement for context-aware identifier renaming, transforming mechanically correct but cryptic output into human-readable code suitable for security analysis workflows.

For fairly and comprehensively evaluate our approach, we curated two dedicated datasets: a) *MalJS*, a large-scale dataset of 23,212 wild malicious JavaScript samples curated from over 10 million ones, covering all known obfuscation techniques. This dataset provides real-world samples with diverse and multi-layer obfuscation techniques, unlike existing datasets with artificial samples. b) *BenignJS* comprises 2,000 manually verified benign samples from GitHub and web crawling. Moreover, we also adopt *CombiBench*, a complex combination obfuscation subset from JsDeObsBench [15], for a fair comparison with existing methods.

Our evaluation methodology encompasses six complementary dimensions to comprehensively assess deobfuscation performance. First, we conduct a deobfuscation capability evaluation across 20 categorized obfuscation techniques. Second, we assess the correctness of deobfuscated code. Third, we evaluate semantic consistency to the original code. Fourth, we evaluate code simplification using complexity metrics from JsDeObsBench. Fifth, we validate effectiveness through entropy-based analysis. Sixth, we conduct multi-LLM readability assessment.

We compare JSIMPLIFIER with 12 existing approaches—including nine traditional tools (JSNice [10], JStillery [8], illuminateJS [9], etc.) and three LLM-based solutions (GPT-o1, Gemini-2.0-Flash, DeepSeek-R1)—selecting appropriate baselines for each evaluation dimension. The results demonstrate strong performance of JSIMPLIFIER across multiple evaluation dimensions. Specifically, our approach demonstrates comprehensive success across all 20 obfuscation techniques with 100% processing capability and correctness on a subset of BenignJS with artificially applied obfuscation. The results maintain semantic consistency with 93.78% Control Flow Graph (CFG) similarity scores and 95.84% Data Dependence Graph (DDG) preservation rates. In addition, our approach achieves a 0.8820 Halstead length reduction in code simplification evaluation on CombiBench, representing an 88.20% reduction in code complexity. This outperforms traditional deobfuscation approaches: JS-deobfuscator [14] scores 0.0015, and Synchrony [13] scores 0.7889.

Moreover, large-scale validation on MalJS demonstrates a comprehensive entropy reduction, while independent LLM-based readability assessment confirms an average improvement of 466.94% across four evaluation models (Claude-3.7-Sonnet, Gemini-2.5-Pro, DeepSeek-R1, GPT-o3). JSIMPLIFIER transforms unreadable obfuscated code into compre-

hensible versions suitable for security analysis workflows. Furthermore, detailed case studies, including analysis of the JSFireTruck [16] campaign, demonstrate JSIMPLIFIER's practical effectiveness in real-world threat analysis contexts where traditional tools fail to provide adequate support.

We summarize our main contributions as follows:

- We present JSIMPLIFIER, a comprehensive JavaScript deobfuscation tool that integrates code normalization, dual-strategy deobfuscation (static AST analysis and controlled execution), and LLM-based semantic enhancement for robust and readable code recovery.
- We demonstrate JSIMPLIFIER's strong performance with 100% processing capability across all obfuscation techniques, exceptional code simplification (0.8820 Halstead reduction), and significant readability improvements validated by multiple LLMs.
- We construct and release the largest real-world obfuscated JavaScript dataset comprising 44,421 samples. This dataset provides valuable insights into real-world obfuscation practices and enables more effective development and evaluation of deobfuscation methods.

**Open Source.** JSIMPLIFIER and datasets are publicly available at Zenodo and GitHub [17]. Detailed usage instructions are in Appendix J.

## II. BACKGROUND

### A. JavaScript

JavaScript is a widely-used programming language that powers web applications and has expanded to server-side and mobile development. Its dynamic nature and flexible syntax make it attractive to malicious attackers for code obfuscation purposes. The language's dynamic features create inherent security vulnerabilities that facilitate sophisticated obfuscation techniques. JavaScript allows variables to change types during execution, enabling transformations where `var x = 5` can later become `x = "hello"`. Additionally, the language supports runtime code modification through functions like `eval("alert('Hello')")` or dynamic property changes such as `window['alert'] = function(){...}`, providing attackers with powerful tools for code manipulation.

Most critically, JavaScript's loose type system provides powerful obfuscation opportunities through its automatic but inconsistent type conversion rules. The + operator performs string concatenation if either operand is a string (`"5" + 3` becomes `"53"`), while other operators like − always perform numeric operations (`"5" − 3` results in 2). These conversion rules enable complex obfuscation chains where expressions like `+!![]` equal 1: the empty array `[]` is truthy [18], `!![]` becomes boolean True, and + converts the boolean value to number 1. In contrast, `!![] + []` equals `"true"` because when added to an empty array, JavaScript converts both operands to strings for concatenation.

Attackers exploit these multi-step conversions to hide simple values behind complex expressions. This enables advanced techniques like JSFUCK [19], which uses only six characters

`[]()!+` to encode arbitrary JavaScript logic. Furthermore, JavaScript treats functions as values that can be stored in variables, passed as arguments, and created dynamically. This enables attackers to build complex control flows and use string-based code execution through `eval()` to create layers of encoding that effectively hide malicious intent from static analysis tools.

### B. JavaScript Obfuscation

We systematically examined JavaScript obfuscation techniques through comprehensive analysis of both academic literature and practical implementations. Our investigation included 12 mainstream obfuscation tools and a literature survey on existing academic research on JavaScript obfuscation techniques [20]–[28]. Additionally, we collected and analyzed real-world obfuscated JavaScript samples from malware datasets (VirusTotal) and commercial obfuscators (Jscrambler [29], Jshaman [30]).

Based on this comprehensive effort, we categorize JavaScript obfuscation techniques according to the fundamental aspects of code they modify. This taxonomy encompasses four main categories covering 20 distinct obfuscation techniques, as detailed in Appendix I.

**Lexical-level Obfuscation.** These techniques transform basic lexical elements while preserving underlying syntactic and semantic structure. Key techniques include identifier renaming (T0), which replaces meaningful variable and function names with random strings like `_0x62ab7d`; indirect property access (T1), which converts `console.log` to `console['\x6c\x6f\x67']`; arithmetization (T2), which replaces simple numbers with complex expressions like `683517 ^ 683398` for 123; string encoding (T3), which encodes strings using Base64 or other schemes; and boolean encoding (T4), which transforms `True` into `!![]`.

**Syntactic-level Obfuscation.** These techniques modify structural organization without changing fundamental logic. This includes assignment-to-function transformations (T6), which wraps simple assignments in immediately invoked function expressions; string reversal (T7), which stores strings backwards and reconstructs them using `split`, `reverse`, and `join`; and advanced encoding methods such as AAEncode (T8), which uses special Unicode characters to encode JavaScript; JJEncode (T9), which creates complex expressions using JavaScript's type coercion; and JSFUCK (T10), which uses only six characters `[]()!+` to represent any JavaScript code, such as encoding the letter `"a"` as `(![]+[])[+!![]]`.

**Semantic-level Obfuscation.** These techniques fundamentally alter implementation logic while maintaining behavioral equivalence. String array techniques (T11, T12) collect all strings into arrays with complex decoding functions using XOR operations. JSON encoding (T13) transforms object literals into encoded strings that are decoded using XOR operations and parsed via `JSON.parse`. Regular expression encoding (T14) converts regex patterns into encoded character arrays that are reconstructed using `new RegExp`. Eval-based transformations (T15) introduce dynamic code generation through

packed and encoded scripts. Control flow flattening (T16) converts linear code into switch-case state machines that execute in seemingly random order. Dead code insertion (T17) adds non-functional code branches and unreachable statements to increase complexity.

**Multi-layer Obfuscation.** These techniques combine multiple obfuscation strategies from different levels to create highly complex transformations. OB obfuscation (T18) implements commercial-grade protection by combining string arrays, control flow manipulation, and anti-debugging measures. LLM-based obfuscation (T19) leverages artificial intelligence to apply multiple techniques intelligently, creating code that appears naturally written but is heavily obfuscated.

### C. JavaScript Obfuscation Tools

Modern JavaScript obfuscation tools implement diverse techniques with varying capabilities. We analyzed twelve representative obfuscation tools: MeSpl (Metasploit Framework JavaScript Obfuscator) [31], JSha (JShaman) [30], JSC (JScrambler) [29], UGL (UglifyJS) [32], OBfu (Obfuscator.io) [33], JSV7 (jsjiami.com.v7) [34], DAFT (DAFT logic Obfuscator) [35], TER (Terser) [36], FRE (FreejsObfuscator) [37], STU (Stunnix JavaScript Obfuscator) [38], GNI (Gnirts Obfuscator) [39], and BEA (BeautifyTools Obfuscator) [40]. Table IX in Appendix A presents a comprehensive comparison of these tools' capabilities across all 20 obfuscation techniques. The analysis reveals significant variation in implementation capabilities across different tools. Commercial tools like JSha achieve the highest technique implementation rates, supporting 16 out of 20 obfuscation categories (80%), but fail to handle common techniques in real-world samples like AAEncode (T8), JJEncode (T9), JSFUCK (T10) and OB obfuscation (T18). However, JSha requires precise configuration tuning to achieve optimal results, creating usability barriers for practitioners. Mid-tier tools such as OBfu and JSC each implement 11 techniques (55%), while specialized tools like MeSpl focus on specific use cases with more limited scope (6 techniques, 30%).

This comprehensive tool analysis reveals that modern obfuscation capabilities pose substantial challenges for security analysis. The widespread availability of powerful obfuscation methods enables attackers to easily access sophisticated techniques, while the diversity of available approaches makes it difficult to develop comprehensive countermeasures. This technological asymmetry between obfuscation and deobfuscation capabilities motivates the need for more advanced reverse engineering approaches.

## III. PROBLEM DEFINITION

### A. A Real-World Motivating Example

The complexity of modern JavaScript obfuscation techniques create substantial challenges for security analysis that existing deobfuscation tools cannot adequately address. To illustrate these challenges, we show a complex obfuscated wild script (Listing 1) from the JSFireTruck campaign [16], which infected 269,552 web pages in just one month.

JSFireTruck represents a particularly sophisticated obfuscation approach that exploits JavaScript's type coercion system to achieve extreme code transformation. It employs only six ASCII characters (`[]()!+`) by leveraging JavaScript's type coercion rules. For example, `+[]` converts empty array to 0, while `!![]` converts to boolean `True` then to number 1. The campaign combines this with `String.fromCharCode` functions and character array lookups, creating multi-layer obfuscation that decodes character arrays, processes JSFireTruck-encoded strings, and executes malicious code.

This sample demonstrates three critical challenges that existing tools cannot address comprehensively.

**Challenge 1: Input Format Diversity.** Existing deobfuscation tools show significant limitations in handling the diverse input formats encountered in real-world malicious samples. Tools like JStillery [8] and IlluminateJS [9] crash when encountering malformed syntax, mixed character encodings, or bundler-wrapped code common in real-world samples. These tools lack robust input validation and normalization capabilities, failing before analysis even begins. This fundamental weakness severely limits their real-world applicability where robust input preprocessing is essential.

```
var _0x4f2a = [/* large character array with
    encoded strings */];
var _0x1b3c = String.fromCharCode(105, 102,
    114, 97, 109, 101...);
var _0x2d4e = String.fromCharCode(115, 114,
    99...);
// 13K characters of obfuscated code follows
[][(![]+[])[+[]]+([![]]+[][[]])
    [+!+[]+[+[]]]+(![]+[])
    [!+[]+!+[]]+(!![]+[])[+[]]+(!![]+[])
    [!+[]+!+[]+!+[]]+(!![]+[])
... // truncated for brevity
```

Listing 1: Snippet of multi-layer obfuscated code.

**Challenge 2: Multi-Layer Obfuscation Complexity.** The JSFireTruck example demonstrates the necessity of combining multiple analysis approaches to achieve comprehensive deobfuscation. The sample combines static patterns (character arrays, `String.fromCharCode` calls) with dynamic type coercion chains that require runtime evaluation. Static-only tools cannot resolve expressions like $+!![] \rightarrow 1$ through complex coercion rules, while dynamic-only approaches struggle with the massive scale (269,552 samples) and security risks of executing malicious code. Furthermore, specialized tools like unjsfuck [41] handle only the JSFUCK layer, leaving intermediate output like `$[0] + $[1].match($[2]|$[3])` that still requires further process.

**Challenge 3: Human Analysis Integration Gap.** Even when tools successfully decode the obfuscation layers, the output commonly contains cryptic identifiers (`_0x4f2a`, `_0x1b3c`) and inconsistent formatting that impedes human analysis. Analysts need semantically meaningful variable names and professional formatting to quickly understand malicious behavior, but existing tools focus solely on functional correct-

ness, ignoring human readability requirements essential for threat intelligence workflows.

*B. Problem Scope*

Our research focuses on "narrow-sense obfuscation" - code transformations that make JavaScript difficult to understand while preserving its original functionality. This includes techniques like identifier renaming, string encoding, control flow flattening, and syntactic transformations that obscure code meaning but keep the code fully executable.

We exclude cryptographic protection schemes that require external decryption keys, as these represent fundamentally different security challenges beyond the scope of traditional deobfuscation. Instead, our approach targets the most common obfuscation techniques used in real-world malware campaigns like JSFireTruck, where the obfuscated code can be executed directly in any JavaScript environment without requiring additional secrets or external resources.

## IV. DESIGN

JSIMPLIFIER adopts a modular approach that systematically combines static AST manipulation, dynamic execution monitoring, and LLM-enhanced processing. The modular design enables robust input handling, comprehensive obfuscation resolution, and human-readable output generation through a unified pipeline, delivering superior deobfuscation capabilities for real-world malware analysis.

*A. Overview*

As Figure 1 shows, JSIMPLIFIER consists of a three-stage pipeline architecture: Preprocessor (①), Deobfuscator (②), and Humanizer (③). Each module tackles a specific aspect of the deobfuscation challenge, collectively enabling effective processing of heavily obfuscated code. This modular design ensures robust handling of complex, multi-technique obfuscation while remaining flexible for future enhancements. The details of each module are described below.

**Preprocessor** (①): The preprocessor validates input code integrity and normalizes syntax through lexical, semantic, and structural transformations. This ensures that subsequent analysis stages receive well-formed, processable code regardless of input variations. As shown in the code transformation example in Figure 1, it handles malformed syntax and prepares the code structure for analysis.

**Deobfuscator** (②): The deobfuscator applies targeted deobfuscation techniques using either static AST pattern matching for well-understood obfuscation patterns or dynamic execution tracing for complex, runtime-dependent transformations. The transformation example in Figure 1 demonstrates how obfuscated constructs are systematically resolved: array-based string encoding (`_0x1 = ['msg']; _0x2 = _0x1[0]`) is simplified through static analysis to direct string assignment (`_0x2 = 'msg'`).

**Humanizer** (③): The humanizer employs LLMs to rename identifiers and improve semantic clarity, transforming mechanically deobfuscated code into human-readable form that
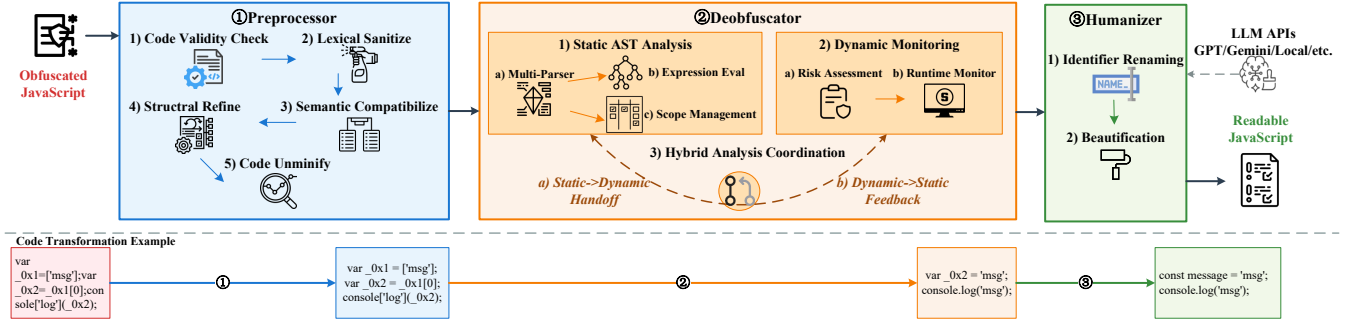
Fig. 1: Overview of JSIMPLIFIER.

facilitates further analysis. As illustrated in the final output in Figure 1, cryptic variable names are replaced with meaningful identifiers. Additionally, industry-standard formatting ensures a consistent professional code presentation that enhances readability and maintains compatibility with development environments.

### B. Preprocessor

The preprocessor module serves as the critical foundation for reliable deobfuscation by ensuring that input code can be properly parsed and analyzed regardless of its initial state. Obfuscated JavaScript often contains malformed syntax, mixed encodings, and compatibility issues that can cause downstream analysis to fail. The preprocessor addresses these fundamental challenges through a systematic normalization pipeline.

*1) Code Validity Check:* We use Meriyah [42] to handle malformed obfuscated JavaScript thanks to its strong fault tolerance and broad syntax support. Unlike traditional parsers that fail on broken or incomplete code, Meriyah generates complete ASTs even with syntax errors or problematic constructs. This capability proves indispensable for processing the diverse range of obfuscated inputs encountered in real-world scenarios where standard parsers would fail entirely.

*2) Lexical Sanitize:* We systematically normalize character encodings through a two-phase transformation process that resolves encoding conflicts in obfuscated JavaScript.

The first phase converts legacy octal escape sequences to standardized hexadecimal format. This ensures parser compatibility across different JavaScript engines, since octal escapes are deprecated in strict mode and inconsistently supported [18]. We use pattern matching to identify three-digit octal patterns (e.g., \302), convert them to decimal values, and reformat them as hexadecimal representations (e.g., \xC2).

The second phase addresses multi-byte UTF-8 character reconstruction. Non-ASCII characters require multiple bytes in UTF-8 encoding (e.g., Chinese characters need three bytes). Obfuscated code often splits these characters into separate \xHH escape sequences that must be recombined for proper decoding. Our state machine parser identifies and groups consecutive escape sequences (e.g., \xE4\xB8\xAD) up to four bytes. We validate hexadecimal pairs and attempt UTF-8 decoding to reconstruct the original characters into standardized

Unicode escapes (e.g., \u4E2D). When UTF-8 reconstruction fails, the system preserves individual hexadecimal escapes to prevent data corruption. Additionally, we extract JavaScript content from HTML containers to ensure consistent input formatting across different JavaScript engines.

*3) Semantic Compatibilize:* We replace legacy JavaScript constructs with cross-platform equivalents to ensure compatibility and strict-mode compliance. Legacy code often includes platform-specific directives and non-standard APIs that break modern parsers.

Our approach handles legacy conditional compilation through pattern recognition and standardization. We identify conditional compilation blocks (e.g., /*@cc_on...@*/) and transform legacy platform detection methods into cross-platform equivalents. For example, legacy @_win32 conditions are converted to navigator.platform checks, and @_jscript detection is replaced with feature detection using 'ActiveXObject' in window. The transformation process also converts legacy conditional syntax (@if, @else, @end) into standard JavaScript control structures.

We address variable declaration inconsistencies through multi-phase processing. The system first identifies explicitly declared variables, then detects implicit global assignments that lack proper declarations, and automatically adds appropriate declaration keywords (let, var) to ensure strict-mode compatibility. This prevents runtime errors in modern JavaScript environments that enforce stricter scoping rules.

For problematic delete statements that violate strict-mode constraints, we implement selective removal based on context analysis. The system identifies delete operations on variables (rather than object properties), removes these invalid statements while preserving surrounding syntax structure, and cleans up residual punctuation to maintain code validity. This ensures that legacy code can execute properly in strict-mode environments without breaking existing program logic.

*4) Structural Refine:* We resolve declaration conflicts and code structure violations through AST scope chain traversal and systematic refactoring. Our implementation detects variable and function redeclarations by analyzing scope hierarchies, then refactors them into strict-mode compliant forms while preserving semantic meaning. This optimization

improves code maintainability and prepares it for effective deobfuscation by eliminating redundant declarations and standardizing code structure.

*5) Code Unminify:* We integrate WebCrack [43] to handle modern web applications that layer multiple obfuscation techniques on top of bundler-generated code. Our system automatically detects bundler-specific patterns (webpack [44], browserify [45]) and applies specialized unpacking algorithms before general deobfuscation techniques. This preprocessing step is essential because attempting to deobfuscate webpack or browserify output without proper unpacking typically yields poor results due to nested obfuscation complexity.

*C. Deobfuscator*

We design a dual-strategy deobfuscation engine combining enhanced static AST analysis with controlled dynamic execution monitoring. Static analysis handles predictable obfuscation patterns, while dynamic analysis captures runtime-generated code through sandboxed execution. This hybrid approach provides comprehensive support across different obfuscation complexity levels.

*1) Static AST Analysis Deobfuscation:* Our static analysis approach introduces a comprehensive pattern recognition and transformation system that systematically handles diverse obfuscation techniques through enhanced AST manipulation. Unlike existing tools that focus on specific obfuscation patterns, our method provides unified handling of lexical, syntactic, and semantic-level obfuscations through enhanced expression evaluation, robust parser integration, and intelligent scope management. We implement these capabilities starting from JStillery's pattern recognition base:

**a) Multi-Parser for Robust AST Construction.** We implement a fault-tolerant parsing system that integrates multiple JavaScript parsers (Esprima [46], Acorn [47], Babel [48]) with automatic fallback mechanisms. When primary parsers encounter syntax errors or malformed code structures common in obfuscated JavaScript, the system automatically switches to alternative parsers with different error tolerance levels. This layered approach ensures successful AST construction even with heavily mangled input code.

**b) Enhanced Expression Evaluation Engine.** We redesign the expression evaluation component to handle sophisticated obfuscation constructs. Specifically, for `LogicalExpressions`, obfuscators frequently use nested `&&` and `||` chains to hide control flow logic. Our evaluation engine implements proper short-circuit evaluation that processes these operators left-to-right, stopping evaluation when the result is determined (e.g., `False && anything` always returns `False`). This enables us to resolve complex boolean chains like `a && b && c || d` into their actual boolean values during static analysis.

Moreover, modern obfuscators exploit ES6 destructuring assignments to obscure variable bindings. For example, instead of simple assignments like `var a = 1; var b = 2;`, obfuscators generate complex patterns such as `[a, b, c] = [getValue(), obj.prop, func.call(this)]`

or nested destructuring like `[x, [y, z]] = [arr[0], [obj.method(), computed()]]`. These patterns break traditional variable tracking because the assignment targets are not immediately visible. We extend `AssignmentExpression` processing to handle `ArrayPattern` destructuring by parsing the left-hand pattern structure, recursively traversing nested array patterns, mapping each element position to its corresponding right-hand value, and creating individual variable bindings for each destructured element while preserving the evaluation order.

For `UnaryExpressions`, obfuscators often apply `typeof` or negation operators to global variables in conditional checks that depend on runtime environment detection. Common obfuscation patterns include `typeof window !== 'undefined' && window.document` for browser detection, `!!(typeof global === 'object' && global)` for Node.js environment checks, or `typeof navigator !== 'undefined' && navigator.userAgent` for user agent sniffing. Static evaluation of these expressions would break environment-specific code paths. We implement an `excludedNames` whitelist containing critical globals (`window`, `document`, `navigator`, `global`, `process`) that should never be statically evaluated. When processing expressions like `typeof window` or `!document`, the system preserves these expressions unchanged to maintain correct runtime behavior across different JavaScript environments.

**c) Intelligent Scope and Dependency Management.** Obfuscated code breaks variable resolution through indirect function calls and complex reference chains that standard scope analysis cannot follow. We address this through two integrated mechanisms: dependency tracking for indirect function calls and enhanced scope chain management for variable resolution.

Our `functionneed` mechanism records dependency relationships when encountering calls where the callee is determined by variable lookup or property access (e.g., `obj[funcName](args)`), ensuring proper resolution before evaluation. Meanwhile, our scope traversal algorithm maintains a scope chain that tracks variable bindings across function boundaries, preserving parent scope access while handling variable shadowing to prevent incorrect substitution that could change program semantics.

*2) Dynamic Monitoring Execution Deobfuscation:* For obfuscation techniques that resist static analysis, we implement a controlled execution monitoring system that safely captures runtime behavior while maintaining security isolation, inspired by dynamic malware analysis approaches [49]. Our approach operates through two integrated phases: pre-execution risk assessment and controlled runtime monitoring.

**a) Pre-Execution Risk Assessment.** Before attempting dynamic execution, we analyze code patterns to identify potential execution hazards. The safety guard system scans for dangerous keyword combinations (`push`, `shift`, `eval`, `await`) that indicate complex nested obfuscation schemes prone to infinite loops or recursive execution deadlocks. This pattern-based risk assessment determines whether code segments can

be safely executed or require alternative handling strategies.

We enhance risk assessment by mapping function dependencies to trace calls between obfuscated functions. This identifies critical functions needing dynamic resolution and their relationships, enabling understanding of execution flows and prioritizing order to avoid circular dependencies.

**b) Controlled Runtime Monitoring.** We use Node.js `vm.runInNewContext` to create isolated execution contexts. Each obfuscated code segment runs in a separate sandboxed VM instance with no access to the file system, network, or global objects. The environment exposes only essential built-in objects while blocking dangerous APIs like `require` and `process`.

We execute obfuscated code segments in the isolated environment and capture their final execution results. The sandbox execution allows us to safely evaluate expressions that cannot be statically analyzed, such as dynamically constructed strings or runtime-dependent calculations. For example, when processing string array deobfuscation, we execute the array construction and access operations to obtain the actual string values. We implement comprehensive security mechanisms including execution timeouts to prevent hanging processes, recursion depth limits to prevent infinite loops, and memory monitoring to prevent resource exhaustion attacks. When runtime execution encounters unresolved references, the system consults static scope analysis results to inject appropriate variable definitions. Our system primarily executes limited-scope code patterns (pure function calls, IIFE executions, object method calls) to minimize exposure to malicious operations, though we recommend additional isolation for production environments.

*3) Hybrid Analysis Coordination:* We implement coordination mechanisms to integrate static and dynamic analysis results through bidirectional information flow.

**a) Static-to-Dynamic Handoff.** When static analysis encounters expressions that cannot be safely evaluated, JSIMPLI-FIER's `canbetransformed` marking mechanism identifies these expressions and packages their context information for dynamic execution. `CallExpression` requiring dynamic resolution typically involves: (1) function calls where the callee is determined by variable lookup or property access (e.g., `objfuncName`), (2) calls to functions that perform runtime code generation or string manipulation that affects program structure, and (3) calls that depend on runtime state or external context not available during static analysis. The system transfers the `CallExpression` context, scope information, and dependency relationships to the dynamic execution monitor, providing necessary context for safe runtime analysis.

**b) Dynamic-to-Static Feedback Integration.** Dynamic execution results are validated and integrated back into the static AST through type-aware processing that maintains semantic correctness. Simple data types (strings, numbers, booleans) are directly converted to literal AST nodes, while function results are parsed back into proper `FunctionExpression` nodes. Complex objects undergo JSON serialization before integration to ensure safe representation in the AST structure.

The system maintains AST structural integrity through careful node replacement that preserves surrounding syntactic context and control flow relationships. Additionally, the system updates variable bindings in the scope chain, replaces complex expressions with their resolved literal values, and triggers re-analysis of dependent code sections that may now be statically evaluable with the new information.

### D. Humanizer

While the Deobfuscator successfully restores program logic and control flow, the resulting code often retains cryptic identifiers and inconsistent formatting that impede human analysis. To address this limitation, we implement a post-processing pipeline that transforms mechanically correct but unreadable code into professionally formatted, semantically clear representations suitable for security analysis and reverse engineering workflows.

*1) Identifier Renaming:* We integrate LLM capabilities for intelligent variable and function renaming that replaces meaningless obfuscated identifiers with semantically meaningful names. Our implementation leverages the core AST-based renaming approach from HumanifyJS [50], adapting it with multi-provider LLM support and optimized context management for deobfuscation workflows.

*2) Beautification:* We integrate Prettier [51] formatting capabilities to ensure deobfuscated output meets professional development standards. The formatting pipeline applies consistent indentation schemes, standardized bracket placement, normalized quote usage, and appropriate semicolon handling according to established JavaScript coding conventions. This transformation ensures that the final output integrates seamlessly with development workflows and maintains compatibility with standard code analysis tools used in security research environments.

## V. EVALUATION

Our evaluation encompasses eight complementary assessment dimensions to comprehensively measure deobfuscation performance, covering deobfuscation capability, result correctness, semantic consistency, code simplification, ablation study, large-scale effectiveness and performance, LLM-based realiability and readability validation and user study.

### A. Dataset

Our evaluation employs three complementary datasets to comprehensively assess deobfuscation performance across different threat scenarios: two datasets (MalJS, BenignJS) curated by us and one dataset (CombiBench) from JsDeobsBench [15]. Table I provides an overview of the datasets.

TABLE I: Evaluation datasets overview.

| Dataset | # Sample | Avg. Size | Source |
|---|---|---|---|
| **MalJS** | 23,212 | 391.78 KB | A corporate partner |
| **BenignJS** | 21,209 | 41.40 KB | GitHub + Web |
| **CombiBench** | 1,296 | 8.06 KB | JsDeobsBench |

*1) Obfuscation Scoring and Sample Selection.:* To ensure dataset's quality, we implemented a rigorous preprocessing pipeline. Inspired by previous work [52], we applied obfuscation scoring methodology to quantify obfuscation levels in our dataset. Following the four-category classification in Section II-B, we assign scores of 1, 2, 3, and 4 points to lexical-level, syntactic-level, semantic-level, and multi-layer obfuscation techniques respectively, with each technique type contributing only once per script.

*2) Malicious JavaScript Dataset (MalJS):* We collected 10,664,628 wild malicious JavaScript samples from July 3, 2019 to July 9, 2025 with the help of a leading cybersecurity company. We first utilized the Meriyah JavaScript parser for syntax validation, filtering out corrupted files and non-JavaScript content. This step retained 8,451,998 syntax-valid samples. To address highly similar samples within malware families, we performed content-based deduplication by normalizing scripts (replacing all string literals with placeholders, removing whitespace and newlines) and removing samples with identical normalized content, resulting in 4,470,565 unique samples. The dataset shows diverse obfuscation complexity with samples using 0-16 techniques (average 8.32). For representative evaluation, we selected samples across the full complexity spectrum (scores 0-35): 700 samples each for scores 1-33, and all available samples for the highest complexity levels (3 and 109 samples for score 34 and 35 respectively), totaling 23,212 samples.

*3) Benign JavaScript Dataset (BenignJS):* This dataset contains 21,209 benign samples from two sources. First, we retained 19,209 samples sampled from the Top 1K popular websites from Tranco [53] with obfuscation scores $\geqslant 1$. Second, we curated 2,000 clean samples verified to be unobfuscated. These include 1,000 samples from popular open-source JavaScript projects on GitHub (trending repositories with over 3,000 stars) collected between April 15 and May 1, 2025, and 1,000 from PublicWWW [54] crawling of legitimate websites. All clean samples were manually checked for absence of obfuscation. For evaluation in Sections V-D and V-E, we apply systematic obfuscation to these clean samples across four categories (500 samples each). Each sample is obfuscated using randomly selected techniques from its category, with at least one and up to all available techniques applied, then processed through JSIMPLIFIER's three-stage pipeline.

*4) Benchmark Evaluation Dataset (CombiBench):* To ensure a fair comparison with existing deobfuscation methods, we also evaluate our approach on the JsDeObsBench benchmark, which is an evaluation framework specifically designed for JavaScript deobfuscation research. We use the complex combination obfuscation subset, termed *CombiBench*, containing 1,296 samples that combine multiple obfuscation techniques from different categories to represent the most challenging evaluation scenarios.

### B. Experimental Setup

Complete experimental configurations and software dependencies are detailed in Appendix J. We employed Joern v1.1.1548 for code property graph generation with Neo4j 4.4.12 (8 GB heap allocation). Additional software included custom JavaScript parsers built on Esprima v4.0.1 for AST manipulation.

For deobfuscated capability evaluation, we applied each of the 20 obfuscation techniques from our categorization (T0-T19) to the test code, generating 20 distinct obfuscated versions. These obfuscated samples were then processed by:

- *Ten state-of-the-art deobfuscation tools*: JSNice (JSN) [10], Dev-coco (DEV) [55], JStillery (JST) [8], illuminateJS (ILL) [9], JS-Deob (JSD) [56], JSbeautifier.org (JSB) [12], synchrony (SYN) [13], JSDE (JSDec) [57], JS-deobfuscator (SDS) [14], and JSIMPO (JSI) [58]
- *Three LLM-based solutions*: GPT-o1 (GPT), Gemini-2.0-Flash (GEM), and DeepSeek-R1 (DSR)
- *Our tool*: JSIMPLIFIER

For LLM integration, we utilized multiple state-of-the-art models via a unified API service: GPT-o3, DeepSeek-R1, Gemini 2.5 Pro, and Claude 3.7 Sonnet. Model-specific configurations included temperature settings of 0.7 for GPT-o3 and 0.3 for other models, with maximum token limits set to 4,000 across all models. API requests were implemented using asynchronous processing with aiohttp [59] for efficiency, incorporating exponential backoff retry mechanisms and proper error handling to ensure robust and reproducible results.

### C. Deobfuscation Capability

To evaluate the deobfuscation capabilities of existing tools, we conducted extensive experiments using the categorized obfuscation techniques. We selected a representative JavaScript code snippet (in Appendix I) and applied all 20 obfuscation techniques individually to generate test cases. This code snippet encompasses various JavaScript constructs including function declarations, variable assignments, string operations, arithmetic expressions, boolean logic, and conditional statements, making it suitable for testing different obfuscation techniques. Additionally, we developed comprehensive demonstration examples for each obfuscation technique to ensure thorough evaluation. The prompt used for LLM-based deobfuscation is provided in Appendix C-A.

**Results.** Table II presents the key comparison results for all evaluated tools. Our analysis reveals significant variations in deobfuscation capabilities across different tool categories and obfuscation techniques, with multi-layered techniques like LLM obfuscation and commercial OB tools posing the greatest challenges for existing deobfuscation solutions.

Traditional tools demonstrate fundamental limitations across obfuscation categories. Tools like JSN and DEV struggle with lexical transformations(T0) and fail on advanced syntactic transformations like expression-to-function conversion. More sophisticated tools (JST, ILL) handle syntactic techniques moderately but face challenges with semantic-level obfuscation. JSI shows specialized but limited capabilities, reliably handling control-flow flattening (T16) and OB obfuscation (T18) but failing on most other techniques due to strict

TABLE II: Comprehensive comparison of deobfuscation capability of different tools.

| Type | Method | JSN | DEV | JST | ILL | JSD | JSB | SYN | JSDec | SDS | JSI | GEM | GPT | DSR | Ours |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Lexical | T0:Rename | | | | | | | | | | | ✓ | ✓ | ✓ | ✓ |
| | T1:Indirect | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ |
| | T2:Arithmetize | | ✓ | ✓ | ✓ | | | | | | | ✓ | ✓ | ○ | ✓ |
| | T3:StringEncode | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ |
| | T4:BooleanEncode | | ✓ | ✓ | ✓ | | | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ |
| Syntactic | T5:Expr2Function | | | ✓ | ✓ | | | | | | ○ | ✓ | ✓ | ✓ | ✓ |
| | T6:Assign2Function | | | ✓ | ✓ | | | | ✓ | ✓ | ○ | ✓ | ✓ | ✓ | ✓ |
| | T7:Reverse | | | | ✓ | | | | | ✓ | | ✓ | ✓ | ✓ | ✓ |
| | T8:AAEncode | | | ✓ | ✓ | | | | | | | ○ | ○ | | ✓ |
| | T9:JJEncode | | | | | | | | | | | ○ | | ○ | ✓ |
| | T10:JSFUCK | | | | | | | | | | | ○ | | ○ | ✓ |
| Semantic | T11:Arrayize | ○ | ○ | | ✓ | | ○ | | | ✓ | | ✓ | ✓ | ✓ | ✓ |
| | T12:StrArrEncode | | ✓ | ○ | ✓ | | | | | ✓ | ○ | ✓ | ✓ | ✓ | ✓ |
| | T13:JSONEncode | | | ○ | | | | | | | ○ | ✓ | ✓ | ○ | ✓ |
| | T14:RegexpEncode | | | ○ | | | | | | | ○ | ✓ | ✓ | ✓ | ✓ |
| | T15:Eval | | ✓ | ○ | ○ | | | | ✓ | | ○ | ✓ | ✓ | ✓ | ✓ |
| | T16:Flattern | ○ | ✓ | ○ | ○ | ✓ | | | | ✓ | | ✓ | ✓ | ✓ | ✓ |
| | T17:Insert | ○ | | ○ | | | | | ✓ | ○ | | | | | ✓ |
| Multi-Lavered | T18:OB | | | | | | | | ○ | ○ | ✓ | ○ | ○ | ○ | ✓ |
| | T19:LLM | ○ | ○ | ✓ | ✓ | ○ | ○ | ○ | ✓ | | | ✓ | ✓ | ✓ | ✓ |

Note: ✓ = fully successful, ○ = partially successful, blank = fully unsupported.

pattern matching requirements. Exotic encoding techniques (T8, T9, T10) and advanced semantic transformations like JSON encoding (T13) challenge traditional tools due to limited semantic analysis capabilities. Insertion technique (T17) and OB obfuscation (T18) represent the most difficult categories, with most existing tools showing complete failure.

LLM-based solutions show superior performance compared to traditional tools. GPT excels most lexical and syntactic techniques while showing strong capabilities in semantic transformations. However, even these advanced approaches struggle with the most sophisticated obfuscation methods.

In contrast, JSIMPLIFIER achieves success across all 20 obfuscation techniques, including the most challenging T17 (insertion) and T18 (OB) techniques. This comprehensive evaluation highlights the significant limitations of existing deobfuscation approaches and demonstrates the need for solutions capable of handling modern obfuscation complexity.

### D. Result Correctness

We evaluate result correctness by assessing whether deobfuscated code maintains syntactic validity with the original code. We conduct automated parser validation on all BenignJS and MalJS samples, and manual verification on 100 BenignJS and 300 MalJS samples for real-world effectiveness validation. Based on their deobfuscation capability (Section V-C), we selected six best-performing methods for comparison: JST, ILL, SDS, GPT, GEM, and DSR.

Manual verification involves: (1) reviewing parser error messages to categorize failure types (e.g., missing brackets, invalid identifiers); (2) examining deobfuscated code structure to identify syntactic malformations that may not trigger parser errors; (3) validating ECMAScript compliance. Tools that fail to process samples or simply output the input content with minor formatting changes are considered unsuccessful. Samples pass only when both automated parser validation and manual verification succeed.

**Results.** Table III presents the correctness evaluation results across 400 samples (100 BenignJS + 300 MalJS) for detailed tool comparison. JSIMPLIFIER achieved 100% syntactic correctness across both BenignJS and MalJS datasets, with zero parsing errors confirmed through manual verification. Traditional tools showed different results: SDS processed 304 samples with full correctness, while JST and ILL processed fewer samples (220 and 164) but maintained high accuracy. LLM-based methods had mixed performance: GEM processed 396 samples but only 180 were correct, while GPT and DSR processed 212 and 160 samples respectively.

Manual verification of syntactically valid outputs confirmed JSIMPLIFIER's 100% result correctness across both datasets. Traditional tools maintained high accuracy on successfully processed samples: SDS achieved 100% (76/76), ILL maintained 100% (41/41), and JST reached 85.45% (47/55). LLM-based approaches showed lower performance: DSR obtained 87.50% (35/40), GPT achieved 66.04% (35/53), and GEM reached 45.45% (45/99).

TABLE III: Correctness and consistency evaluation results.

| Metrics | JST | ILL | SDS | GPT | GEM | DSR | Ours |
|---|---|---|---|---|---|---|---|
| #Deob (/400) | 220 | 164 | 304 | 212 | 396 | 160 | **400** |
| #Correct(/400) | 188 | 164 | 304 | 140 | 180 | 140 | **400** |
| #Consist (/100) | 26 | 20 | 40 | 26 | 25 | 19 | **83** |
| CFG Sim. (%) | 58.06 | 74.95 | 67.04 | 58.35 | 76.75 | 92.75 | **93.78** |
| DDG Pres. (%) | 59.00 | 79.20 | 63.52 | 55.50 | 78.50 | 89.50 | **95.84** |

### E. Semantic Consistency

We verify that deobfuscated code maintains identical runtime behavior and logical flow compared to original samples using automated graph-based analysis with manual verification. Specifically, Joern generates CFGs and DDGs for structural comparison. The evaluation employs a two-tier approach: for graphs with ≤40 nodes, the Maximum Common

Subgraph (MCS) algorithm computes similarity as the ratio of MCS size to maximum node count. For larger graphs (>40 nodes), fast approximation based on node/edge count differences avoids exponential complexity. When similarity scores fall below 75%, we conduct manual analysis. Samples are marked consistent only when manual verification confirms preserved program behaviors.

**Results.** Table III presents the semantic consistency evaluation results. JSIMPLIFIER achieved 83% semantic consistency (83 out of 100 samples), outperforming GPT (74.29%, 26/35 samples) and JST (55.32%, 26/47 samples). Additionally, structural analysis shows JSIMPLIFIER maintains superior code fidelity with 93.78% CFG similarity and 95.84% DDG preservation, compared to GPT's 58.35% and 55.50%.

To verify this high consistency rate, we applied a manual verification process with specific criteria. For the 83 consistent samples, we confirmed: (1) Identical execution paths with conditional branches, loops, and function calls following the same logical sequence; (2) Preserved variable relationships including assignments, scoping, and data dependencies; (3) Equivalent API usage maintaining identical parameters and invocation order, allowing syntactic variations like `obj["prop"]` vs `obj.prop`; (4) Consistent computational results. Appendix B shows a representative consistent sample[1] demonstrating preserved control flow and data dependencies.

**Failure Analysis.** To address the 17% semantic inconsistency cases, we conducted failure analysis. For the 17 inconsistent samples, our analysis revealed: (1) Altered control flow (8 samples): missing or additional conditional branches, modified loop conditions, or changed function call sequences; (2) Broken variable dependencies (6 samples): incorrect variable scoping, missing assignments, or spurious data relationships; (3) Modified API behavior (4 samples): changed function parameters, altered invocation order, or missing critical API calls; (4) Computational discrepancies (3 samples): different mathematical results, incorrect string operations, or modified logical outcomes. Some samples exhibited multiple inconsistency types, with 4 samples showing overlapping issues. The primary cause stems from LLM-based variable renaming introducing semantic errors. When the LLM performs automatic variable renaming, it may inadequately analyze dynamic variable assignments and scoping, leading to undefined variables that break the original call chains and functionality. A detailed failure case is provided in Appendix H.

### F. Code Simplification

As in previous work [15], we adopt the Halstead length [60] to measure code complexity and the Halstead Length Reduction (HLR) score to evaluate the effectiveness of deobfuscation tools. The HLR score is calculated as $(HLoC_{obf} - HLoC_{deobf})/HLoC_{obf}$, where higher values indicate greater complexity reduction. According to the definition, the HLR score ranges from 0 to 1, where higher values indicate greater

complexity reduction. We also evaluate Halstead Effort Reduction (HER) score, which measures the reduction in mental effort required to understand code. The evaluation uses the CombiBench dataset, comparing against SDS and SYN as the primary baseline tools evaluated in the JsDeObsBench.

**Results.** Table IV presents the complexity reduction scores following the JsDeObsBench evaluation setup. We achieved an HLR score of 0.8820 and HER of 0.9291, demonstrating substantial complexity reduction capabilities. SDS [14] achieved minimal simplification with HLR and HER score of only 0.0015 and 0.1174. While SYN [13] showed better performance with HLR and HER score of 0.7889 and 0.8736, it can only handle specific obfuscation combinations and lacks generalization to diverse real-world obfuscation patterns. Our results demonstrate that JSIMPLIFIER achieves substantial complexity reduction, confirming our method's effectiveness in transforming obfuscated code into obvious forms.

TABLE IV: Results of HLR and HER score comparison.

| Tool | HLR Score | HER Score |
|---|---|---|
| JSIMPLIFIER | **0.8820** | **0.9291** |
| synchrony (SYN) | 0.7889 | 0.8736 |
| JS-deobfuscator (SDS) | 0.0015 | 0.1174 |

### G. Ablation Study

To quantify each module's contribution, we evaluated four configurations on 1,000 randomly selected MalJS samples (1KB to 5.49MB, average 82.23KB): (1) Preprocessor only, (2) Preprocessor + Static Analysis, (3) Preprocessor + Deobfuscator (Static + Dynamic), and (4) Full system (with Humanizer).

**Results.** Table V shows the ablation results. Processing times remain similar across most configurations (9.16-9.26 seconds) due to fixed-cost operations like file I/O and AST parsing. The incremental overhead from static and dynamic modules is relatively small since most samples do not trigger complex obfuscation-specific processing. The full system requires 87.31 seconds due to LLM API latency but achieves optimal deobfuscation quality.

Table VI presents detailed deobfuscation capability breakdown across all 20 obfuscation techniques. The Preprocessor handles 0/20 techniques (syntactic normalization only). Static analysis enables 14/20 techniques. The combined Preprocessor + Deobfuscator (static + dynamic) handles 19/20 techniques, missing only identifier renaming. The full system achieves complete 20/20 coverage.

For code simplification, we measured Halstead Length Reduction (HLR) on real-world samples. Static analysis achieves 3.52% HLR. The combined static+dynamic approach achieves 3.40% HLR (slightly lower due to additional code introduced during dynamic analysis). The full system achieves 5.66% HLR, significantly lower than the 88.20% achieved on CombiBench (Section V-F) due to real-world complexity.

### H. Large-Scale Effectiveness and Performance

To validate practical code simplification effectiveness and processing efficiency, we conducted large-scale evaluation

---

[1]MD5: 000c5123cd553d14cd49d22022fcc9c5

TABLE V: Ablation study showing module contributions.

| Configuration | Avg. Time (s) | Techniques | HLR (%) |
|---|---|---|---|
| Preprocessor | 9.26 | 0/20 | 0.00 |
| + Static | 9.16 | 14/20 | 3.52 |
| + Dynamic | 9.23 | 19/20 | 3.40 |
| + LLM | 87.31 | **20/20** | **5.66** |

TABLE VI: Deobfuscation capability by module configuration.

| Technique | Prep | +Static | +Dynamic | +LLM |
|---|---|---|---|---|
| T0 | | | | ✓* |
| T1-T8 | | ✓ | ✓ | ✓ |
| T9-T10 | | | ✓* | ✓ |
| T11-T14 | | ✓ | ✓ | ✓ |
| T15-T16 | | | ✓* | ✓ |
| T17 | | ✓ | ✓ | ✓ |
| T18 | | | ✓* | ✓ |
| T19 | | ✓ | ✓ | ✓ |
| **Total** | **0/20** | **14/20** | **19/20** | **20/20** |

* indicates newly enabled capability in this configuration.



Fig. 2: Entropy reduction across (a) MalJS and (b) BenignJS.

TABLE VII: Performance comparison across MalJS dataset.

| Tool | Success | Time (s) | Memory (MB) | Max File |
|---|---|---|---|---|
| JST | 45.67% | 22.69 | **+0.0056** | 17.51MB |
| ILL | 19.06% | **6.26** | +0.3130 | 7.58KB |
| JSIMPLIFIER | **100%** | 83.17 | +0.1034 | **29.58MB** |

on MalJS (23,212 samples) and BenignJS (21,209 samples) datasets. We performed full-scale entropy analysis on both datasets to assess effectiveness, and comprehensive performance evaluation on the complete MalJS dataset.

We employed two complementary entropy metrics to quantify deobfuscation quality: Code Text Entropy measures textual randomness through character and word frequency analysis, while AST Entropy evaluates structural complexity using node types, connectivity, depth, and edge relationships. We selected JST and ILL as comparison baselines due to their deobfuscation capabilities and batch processing support. For fair comparison, we measured only JSIMPLIFIER's Preprocessor and Deobfuscator, excluding the Humanizer. Detailed formulations are provided in Appendix E.

**Effectiveness Results.** JSIMPLIFIER successfully processed all 44,421 samples (100%), while JST processed 30,188 samples (67.9%) and ILL handled 14,140 samples (31.8%). This comprehensive coverage is essential for practical malware analysis. Figure 2 shows JSIMPLIFIER achieved the lowest median entropy values for both AST structure and code text across both datasets. The consistent reduction patterns between malicious and benign samples confirm our approach generalizes well beyond malware-specific obfuscation. The dual entropy reduction aligns with our HLR scores, confirming comprehensive deobfuscation capability.

**Performance Results.** Table VII presents performance metrics showing JSIMPLIFIER's 100% success rate versus JST's 45.67% and ILL's 19.06%, with processing times of 83.17 seconds that are acceptable for malware analysis scenarios. JSIMPLIFIER maintains consistent performance across all file sizes including very large files ($>10MB$) with moderate memory consumption (0.1034 MB average increment). In contrast, JST exhibits low memory usage (0.0056 MB) but poor success rate, while ILL's fixed allocation (0.3130 MB) limits capability to 7.58KB maximum file size, causing baseline tools to frequently timeout on complex samples.
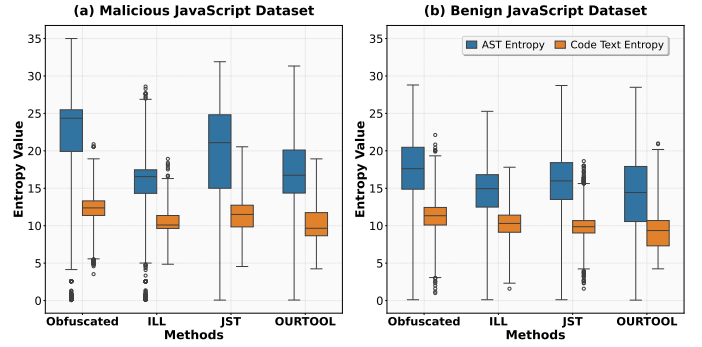
### I. LLM-based Realiability and Readability

To evaluate practical effectiveness, we assess code readability improvement and processing reliability through LLM-based evaluation, avoiding human assessment subjectivity. For readability assessment, we selected 60 sample pairs from MalJS for cross-model validation using Claude 3.7 Sonnet, Gemini 2.5 Pro, DeepSeek-R1, and GPT-o3. Each LLM scored obfuscated and deobfuscated code pairs on a 0-10 readability scale using customized prompts (Appendix C-B). For reliability analysis, we selected 100 samples from Section V-G for identifier renaming using GPT-4o-mini.

**Readability Results.** Table VIII shows substantial readability improvement across all evaluators. JSIMPLIFIER achieves 466.94% average improvement, transforming unreadable obfuscated code (scores 1.02-1.81) into comprehensible versions (scores 6.21-7.83). Cross-model consistency validates improvement objectivity. Deobfuscated scores around 6-8 points indicate practical readability for security analysis, corroborating HLR score 0.8820. Consistent improvements across samples support entropy reduction findings (detailed visualization in Appendix F).

TABLE VIII: LLM-based readability assessment validating complexity reduction.

| Model | Original | Deobfuscated | Improvement |
|---|---|---|---|
| Claude 3.7 Sonnet | 1.46 | 7.12 | +387.67% |
| Gemini 2.5 Pro | 1.02 | 7.33 | +618.63% |
| DeepSeek-R1 | 1.09 | 7.83 | +618.35% |
| GPT-o3 | 1.81 | 6.21 | +243.09% |

**Realiability Results.** The 100-sample GPT-4o-mini evaluation generated 20,897 API calls consuming 6,446,654 tokens, costing $1.04 total with 10.96 hours processing time. Average per-sample cost was $0.0104 with 6.6 minutes processing time. Long-tail distribution showed median cost $0.0003 and median time 10.7 seconds. Samples ranged from 250 bytes to 2.44 MB

(average 80 KB, median 8 KB). Manual inspection revealed a 12% hallucination rate, defined as identifier renaming deviating from ground truth expectations. Cases involved overly verbose naming (e.g., `i` to `iterationController`) reflecting over-interpretation. The remaining 88% demonstrated reliable renaming.

### J. User Study

To complement objective LLM-based assessment with human validation of real-world usability, we conducted a user study with 9 participants across three expertise levels: novice (3 participants: undergraduates and developers), intermediate (3 participants: graduate students and CTF players), and expert (3 participants: security researchers and analysts). Each participant analyzed 6 obfuscated JavaScript samples under three conditions: Original (raw obfuscated), Traditional (processed by JST), and JSIMPLIFIER. The samples were balanced across complexity levels and malicious behavior types using a Latin square design [61] to control for learning effects and individual differences. Participants completed standardized analysis tasks (functionality identification, variable recognition, execution flow ordering, threat classification, IOC extraction) and provided subjective evaluations on 5-point Likert scales [62] for readability, clarity, logicality, difficulty, and confidence. The complete questionnaire is provided in Appendix G.

**Results.** Table X (Appendix G) shows JSIMPLIFIER consistently improved analysis accuracy across all groups. Novices gained the most (+12.7%). Analysis time decreased substantially, especially for intermediates (-47.7%). Subjective ratings for readability, clarity, and logicality showed marked improvements, often doubling compared to Original. Higher difficulty and confidence scores indicate JSIMPLIFIER improved code analyzability and boosted analysts' confidence. Participant feedback on a heavily obfuscated jQuery UI sample[2] illustrates practical impact. One intermediate participant noted: *The variable renaming made it possible to follow the logic flow, and I could identify the suspicious network calls within minutes instead of struggling for the entire session.* This aligns with quantitative results showing 50% analysis time reduction and substantial readability improvements.

## VI. DISCUSSION

### A. JSIMPLIFIER *in Security Analysis Workflows*

To further demonstrate JSIMPLIFIER's practical value, we evaluated its performance on the JSFireTruck campaign sample that motivated this research (Listing 1 in Section III-A). This case study reveals how JSIMPLIFIER addresses real-world challenges that existing tools cannot handle.

As shown in Listing 2, JSIMPLIFIER successfully deobfuscates the complete multi-layer obfuscation chain that defeats existing tools. While specialized tools like unjsfuck only handle the basic layer, leaving intermediate output such as `$[0] + $[1].match($[2]|$[3])` requiring manual analysis, JSIMPLIFIER processes all obfuscation layers automatically.

---

[2]Sample MD5: 365df2394d7f7cf6ebdc7dfac6d2b3e6

---

The tool transforms thousands of characters of JSFireTruck-encoded expressions into clear, readable code that directly exposes malicious behavior, such as referrer-based victim filtering, iframe injection with CSS manipulation, and evidence removal. Security analysts can quickly understand attack intent without manual deobfuscation, demonstrating JSIMPLIFIER's practical value for real-world security analysis.

```
if (document.referrer.match(/google|bing|yahoo
    |duckduckgo/)) {
var maliciousIframe = '<iframe src="' +
    maliciousUrl +
'" style="z-index:30000;width:100%;height
    :100%;' +
'position:absolute;top:0;left:0;"></iframe>';
document.getElementById(randomId).innerHTML =
    maliciousIframe;
}
document.currentScript.remove();
```

Listing 2: JSIMPLIFIER's deobfuscated output revealing malicious intent.

### B. Anti-Analysis Technique Handling

Beyond traditional obfuscation, malicious JavaScript employs anti-analysis techniques to evade detection. JSIMPLIFIER handles these through static analysis and dynamic execution.
**Time Bombs.** JSIMPLIFIER combines static analysis for time-based logic simplification with controlled dynamic execution. Listing 4 (Appendix D) shows time bomb handling where time-based conditions are exposed through static simplification, allowing analysts to identify activation timestamps (January 1, 2022 in this example) and assess threat timelines without triggering malicious payloads.
**Environmental Probes.** JSIMPLIFIER handles environment detection through browser API simulation and controlled sandbox execution. Our approach provides DOM-like stubs (`document`, `window`, `navigator`) and whitelists essential browser APIs (`btoa`, `atob`, `setTimeout`) as safe globals, preventing `ReferenceError` exceptions while ensuring environment checks receive plausible responses without exposing the host system. This allows deobfuscation to complete and expose evasion logic for analyst review.

### C. Implications for LLM-based Deobfuscation

Our results have important implications for LLM-based approaches in deobfuscation workflows. While JsDeObsBench demonstrated potential of pure LLM approaches for code simplification and readability improvement, it revealed significant reliability concerns, context window constraints, unpredictable failure modes, and high computational costs that limit practical deployment. Our hybrid approach validates the complementary nature of traditional and LLM-based methods by utilizing static AST analysis and dynamic execution for core deobfuscation tasks while leveraging LLM capabilities for readable output generation. This targeted integration achieves both traditional method reliability and LLM readability improvements (466.94% average improvement with 88% processing reliability at $0.0104 per sample).

## D. Limitations

While JSIMPLIFIER shows a significant advancement in JavaScript deobfuscation capabilities, some limitations remain. **Compatibility Constraints.** To support older syntax, ES6+ features may be downgraded during preprocessing, potentially limiting modern JavaScript capabilities in deobfuscated output. We mitigate this through selective normalization disabling and multi-parser fallback mechanisms.

**LLM Integration Challenges.** Our Humanizer module faces inherent challenges in understanding obfuscated code context and domain-specific terminology. LLMs may generate semantically plausible but contextually inappropriate names, particularly in malware analysis where technical precision is crucial. Context window limitations may prevent full understanding of complex variable relationships across large codebases. We address this through multi-provider LLM architecture for suggestion comparison, validation mechanisms checking generated names against existing scope bindings, and fallback strategies preserving original identifiers when validation fails. A detailed failure case is provided in Appendix H.

**Security and Scope Limitations.** JSIMPLIFIER faces several constraints, including adversarial resistance and scope limitations. Sophisticated adversaries can craft targeted obfuscation to bypass specific techniques, creating an inevitable arms race. As a deobfuscator, we cannot handle cryptographic packing or custom language interpreters that hide code. Our approach also cannot handle time-dependent obfuscation techniques that use runtime values for code decryption. Our dynamic execution using Node.js `vm.runInNewContext` carries theoretical VM escape risks that may require additional containerization.

## VII. RELATED WORK

### A. JavaScript Obfuscation Detection

Large-scale analyses demonstrate obfuscation's widespread adoption [22]–[28]. Moog [22] analyzes Alexa Top 10K scripts using control-flow-enhanced AST classifiers, while Sarker [23] discovers most domains host obfuscated scripts through runtime-static analysis. Jack-in-the-box [24] examines vulnerable npm bundles in top sites. Advanced evasion techniques continue emerging: HideNoSeek [25] camouflages malware in benign ASTs, Wobfuscator [26] embeds logic in WebAssembly, and JSPro [27] reduces runtime overhead via WASM translation. Detection methods include runtime-static discrepancy analysis [23] and FastText with virtual machine-based deobfuscation [28]. While these studies focus on obfuscation detection, our work addresses the complementary challenge of comprehensive deobfuscation.

### B. JavaScript Deobfuscation Techniques

Researchers have developed various deobfuscation approaches to counter script obfuscation [58], [63]–[69]. Most techniques focus on specific obfuscation aspects. JSimpo [58] provides structural deobfuscation, TransAST [63] leverages AST template mappings for identifier recovery, while JS-Naughty [64] uses statistical machine translation and DE-OBFUSCATING [65] employs character-based tokenization

with Seq2Seq models. Eval obfuscation receives targeted solutions from Arceri [66] through abstract interpretation and JSDES [67] via dynamic stubbing. Multi-technique approaches include SAFE-DEOBS [68] with AST-based optimization passes and Hajarnis [69] integrating taint analysis and symbolic execution. Recent LLM evaluations reveal significant limitations: GPT-4 shows poor functional deobfuscation for advanced techniques [70], while JsDeObsBench [15] demonstrates critical reliability issues with average failure rates of 2.76% for syntax correctness and 37.40% for execution correctness. Unlike these fragmented approaches that excel only in narrow domains or suffer from reliability issues, JSIMPLIFIER provides comprehensive deobfuscation through systematic integration of static AST analysis, controlled dynamic execution, and targeted LLM post-processing.

### C. Other Language Deobfuscation Research

Deobfuscation research across languages provides valuable insights [70]–[81]. LLM approaches show mixed results: DOBF [71] outperforms CodeBERT through pre-training, while GPT-4 achieves limited functional deobfuscation [70]. Dynamic methods demonstrate effectiveness through Power-Peeler [72] for PowerShell reconstruction, Oblivion [73] for VBA macro instrumentation, and API-Xray [74] for import reconstruction. Expression simplification techniques include ProMBA [75] for MBA synthesis, MBA-Blast [76] with transformations, and GAMBA [77] for nonlinear MBAs. Language-specific tools target particular environments: StringHound [78] for Android string obfuscation, DeProDexcator [79] for Pro-Guard reversal, while systematic knowledge surveys [80] analyze virtualization deobfuscation approaches, and PHP research [81] explores layout obfuscation with AES encryption. JSIMPLIFIER is purpose-built for JavaScript's unique characteristics to provide native handling of JavaScript-specific obfuscation patterns while maintaining compatibility with the language's execution model and ecosystem requirements.

## VIII. CONCLUSION

This work proposes JSIMPLIFIER, a comprehensive JavaScript deobfuscation tool. JSIMPLIFIER addresses limitations in existing approaches through multi-stage processing and LLM integration. It handles 20 categories of obfuscation techniques using static AST analysis, dynamic execution, and semantic-aware code humanization. We maintains correctness through strict semantic validation and CFG/DDG consistency checking. Experimental results demonstrate JSIMPLIFIER's effectiveness across multiple evaluation dimensions. JSIMPLIFIER successfully processes all samples in evaluation datasets. It maintains 100% correctness on processed samples and achieves 83% semantic consistency, outperforming both traditional and LLM-based approaches. Code simplification evaluation demonstrates an HLR score of 0.8820. LLM-based readability assessment shows 466.94% average improvement. Large-scale evaluation on 44,421 samples demonstrates significant entropy reduction, confirming JSIMPLIFIER's effectiveness for real-world malware analysis.

REFERENCES

[1] K. Ren, W. Qiang, Y. Wu, Y. Zhou, D. Zou, and H. Jin, "An empirical study on the effects of obfuscation on static machine learning-based malicious javascript detectors," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023.

[2] S. Ndichu, S. Kim, and S. Ozawa, "Deobfuscation, unpacking, and decoding of obfuscated malicious javascript for machine learning models detection performance improvement," *CAAI Transactions on Intelligence Technology*, 2020.

[3] L. Gong, "Dynamic analysis for javascript code," Ph.D. dissertation, University of California, Berkeley, 2018.

[4] E. Vilchik, "Static analysis in javascript: What's easy and what's hard," GitNation Conference Talk, 2024, conference presentation, Accessed: Jan. 15, 2025. [Online]. Available: https://gitnation.com/contents/static-analysis-in-javascript-whats-easy-and-whats-hard

[5] Trend Micro Research, "SocGholish's intrusion techniques facilitate distribution of ransomware," Trend Micro Research, 2025, accessed: Mar. 27, 2025. [Online]. Available: https://www.trendmicro.com/en_us/research/25/c/socgholishs-intrusion-techniques-facilitate-distribution-of-rans.html

[6] P. T. R. Team, "Part 1: Socgholish: A very real threat from a very fake update," December 2022, accessed: Jan. 15, 2025. [Online]. Available: https://www.proofpoint.com/us/blog/threat-insight/part-1-socgholish-very-real-threat-very-fake-update

[7] K. Brezinski and K. Ferens, "Metamorphic malware and obfuscation: A survey of techniques, variants, and generation kits," Security and Communication Networks, Wiley, 2023, article ID: 8227751, Accessed: Jan. 15, 2025. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1155/2023/8227751

[8] Minded Security, "JStillery - advanced JavaScript deobfuscation via partial evaluation," GitHub Repository, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://github.com/mindedsecurity/jstillery

[9] Geeks on Security, "IlluminateJS - a static JavaScript deobfuscator," GitHub Repository, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://github.com/geeksonsecurity/illuminatejs

[10] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from big code," in *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2015.

[11] Kasada, "Javascript deobfuscation: Hiding intent to fortify bot defenses," Kasada Security Blog, 2023, accessed: Jan. 15, 2025. [Online]. Available: https://www.kasada.io/javascript-deobsfusction-bot-defenses/

[12] E. Lielmanis and L. Newman, "js-beautify: Beautifier for javascript, html, css," GitHub Repository, 2024, accessed: Jan. 15, 2025. [Online]. Available: https://github.com/beautifier/js-beautify

[13] Relative, "Synchrony: Javascript cleaner & deobfuscator," GitHub Repository, 2024, accessed: Jan. 15, 2025. [Online]. Available: https://github.com/relative/synchrony

[14] ben-sb, "deobfuscator-io-archived: Deobfuscator for obfuscator.io," GitHub Repository, 2024, accessed: Jan. 15, 2025. [Online]. Available: https://github.com/ben-sb/javascript-deobfuscator

[15] G. Chen, X. Jin, and Z. Lin, "Jsdeobsbench: Measuring and benchmarking llms for javascript deobfuscation," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2025.

[16] Unit 42, "JSFireTruck: Exploring malicious JavaScript using JSF*ck as an obfuscation technique," Palo Alto Networks Unit 42, 2025, accessed: Jul. 3, 2025. [Online]. Available: https://unit42.paloaltonetworks.com/malicious-javascript-using-jsfiretruck-as-obfuscation/

[17] XingTuLab, "JSIMPLIFIER – JavaScript code simplification and deobfuscation tool," GitHub Repository, 2025, accessed: Dec, 2025. [Online]. Available: https://github.com/XingTuLab/JSIMPLIFIER

[18] Ecma International, "Ecmascript 2023 language specification," Ecma International, Standard ECMA-262, 2023, accessed: Jan. 15, 2025. [Online]. Available: https://www.ecma-international.org/publications-and-standards/standards/ecma-262/

[19] HandWiki, "JSF*ck," HandWiki Encyclopedia, 2024, accessed: Dec. 19, 2024. [Online]. Available: https://handwiki.org/wiki/JSFuck

[20] P. Skolka, C.-A. Staicu, and M. Pradel, "Anything to hide? studying minified and obfuscated code in the web," in *The World Wide Web Conference (WWW)*, 2019.

[21] W. Xu, F. Zhang, and S. Zhu, "The power of obfuscation techniques in malicious javascript code: A measurement study," in *International Conference on Malicious and Unwanted Software: "The Americas" (MALWARE)*, 2013.

[22] M. Moog, M. Demmel, M. Backes, and A. Fass, "Statically detecting JavaScript obfuscation and minification techniques in the wild," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021.

[23] S. Sarker, J. Jueckstock, and A. Kapravelos, "Hiding in plain site: Detecting JavaScript obfuscation through concealed browser API usage," in *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2020.

[24] J. Rack and C.-A. Staicu, "Jack-in-the-box: An empirical study of JavaScript bundling on the web and its security implications," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.

[25] A. Fass, M. Backes, and B. Stock, "HideNoSeek: Camouflaging malicious JavaScript in benign ASTs," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[26] A. Romano, Y. Zheng, W. Wang, L. Ying, and G. Vigna, "Wobfuscator: Obfuscating JavaScript malware via opportunistic translation to WebAssembly," in *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2022.

[27] S. Wang, P. Wang, and D. Wu, "Leveraging WebAssembly for numerical JavaScript code virtualization," *IEEE Access*, 2019.

[28] S. Ndichu, S. Kim, and S. Ozawa, "Deobfuscation, unpacking, and decoding of obfuscated malicious JavaScript for machine learning models detection performance improvement," *CAAI Transactions on Intelligence Technology*, 2020.

[29] Jscrambler, "Pioneering client-side protection platform," Jscrambler, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://jscrambler.com/

[30] JShaman, "JavaScript obfuscation encryption," Shaman Technology, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://www.jshaman.com/

[31] Metasploit Documentation, "JavaScript obfuscation," Rapid7 Metasploit Documentation, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://docs.metasploit.com/docs/development/developing-modules/libraries/obfuscation/how-to-obfuscate-javascript-in-metasploit.html

[32] Mishoo, "UglifyJS - JavaScript parser / mangler / compressor / beautifier toolkit," GitHub Repository, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://github.com/mishoo/UglifyJS

[33] JavaScript Obfuscator Tool, "A free and efficient obfuscator for JavaScript," obfuscator.io, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://obfuscator.io/

[34] JSJiami, "Online JavaScript code obfuscation and encryption tool," jsjiami.com, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://www.jsjiami.com/

[35] Daft Logic, "Online Javascript obfuscator," Daft Logic, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://www.daftlogic.com/projects/online-javascript-obfuscator.htm

[36] Nodejs.cn, "Terser - JavaScript obfuscator and compressor toolkit," Terser Official Documentation, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://terser.nodejs.cn/

[37] FreeJSObfuscator Team, "Free JavaScript obfuscator - 100% free JS code protection tool," FreeJSObfuscator.com, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://www.freejsobfuscator.com/

[38] Stunnix, "JavaScript obfuscator - protect JavaScript code from illegal reuse and ip theft," Stunnix, 2024, accessed: Jul. 3, 2025. [Online]. Available: http://stunnix.com/prod/jo/

[39] Anseki, "gnirts - obfuscate string literals in JavaScript code," GitHub Pages, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://anseki.github.io/gnirts/

[40] BeautifyTools, "Online Javascript obfuscator - protect JavaScript code from theft," BeautifyTools, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://beautifytools.com/javascript-obfuscator.php

[41] karust, "unjsfuck - JSFuck deobfuscator," GitHub, 2024, accessed: Jul. 4, 2025. [Online]. Available: https://github.com/karust/unjsfuck

[42] Meriyah Team, "Meriyah - a 100% compliant, self-hosted javascript parser," GitHub Repository, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://github.com/meriyah/meriyah

[43] j4k0xb, "webcrack - deobfuscate obfuscator.io, unminify and unpack bundled javascript," GitHub Repository, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://github.com/j4k0xb/webcrack

[44] Webpack Team, "webpack - a static module bundler for modern JavaScript applications," webpack.js.org, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://webpack.js.org/

[45] Browserify Team, "Browserify - browser-side require() the node.js way," browserify.org, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://browserify.org/

[46] A. Hidayat, "Esprima - ECMAScript parsing infrastructure for multipurpose analysis," Esprima.org, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://esprima.org/

[47] Acorn Team, "Acorn - a tiny, fast JavaScript parser, written completely in JavaScript," GitHub Repository, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://github.com/acornjs/acorn

[48] Babel Team, "Babel - JavaScript compiler," Babel Official Website, 2024, accessed: Jul. 4, 2025. [Online]. Available: https://babeljs.io/

[49] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert, "Rozzle: De-cloaking internet malware," ser. SP '12. USA: IEEE Computer Society, 2012.

[50] Jehna, "Humanify - deobfuscate Javascript code using ChatGPT," GitHub Repository, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://github.com/jehna/humanify

[51] Prettier Team, "Prettier - opinionated code formatter," prettier.io, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://prettier.io/docs/

[52] H. Chai, L. Ying, H. Duan, and D. Zha, "Invoke-deobfuscation: Ast-based and semantics-preserving deobfuscation for powershell scripts," in *Proceedings of the 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2022.

[53] Tranco Team, "Tranco: A research-oriented top sites ranking hardened against manipulation," 2024, accessed: 2024-12-19. [Online]. Available: https://tranco-list.eu/

[54] PublicWWW, "PublicWWW - source code search engine," publicwww.com, 2024, accessed: Jul. 3, 2025. [Online]. Available: https://publicwww.com/

[55] dev-coco, "Javascript deobfuscator - online tool," Online Tool, 2024, accessed: Jan. 15, 2025. [Online]. Available: https://dev-coco.github.io/Online-Tools/JavaScript-Deobfuscator.html

[56] kuizuo, "js-deobfuscator: Javascript code deobfuscation tool," GitHub Repository, 2024, accessed: Jan. 15, 2025. [Online]. Available: https://github.com/kuizuo/js-deobfuscator

[57] hax0r31337, "Jsdec: Online javascript decoder," GitHub Repository, 2021, supports sojson v4/Premium/v5. Project archived October 10, 2021, Accessed: Jan. 15, 2025. [Online]. Available: https://github.com/hax0r31337/JSDec

[58] T. Chen, D. Li, Y. Zhang, and T. Xie, "JSimpo: Structural deobfuscation of JavaScript programs," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2025.

[59] aio-libs team, "aiohttp: Asynchronous http client/server framework for asyncio and python," GitHub Repository, 2025, version 3.12, Accessed: Jan. 15, 2025. [Online]. Available: https://github.com/aio-libs/aiohttp

[60] M. H. Halstead, *Elements of Software Science*, ser. Operating and Programming Systems Series. Elsevier Science Inc, 1977.

[61] R. Kirk, *Kirk, Roger E. (2013). Experimental design: Procedures for the behavioral sciences (4th ed.). Thousand Oaks, CA: Sage.*, 06 2013.

[62] R. Likert, "A technique for the measurement of attitudes," *Archives of psychology*, vol. 22, no. 140, p. 55, 1932.

[63] Y. Qin, W. Wang, Z. Chen, H. Song, and S. Zhang, "TransAST: A machine translation-based approach for obfuscated malicious JavaScript detection," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2023.

[64] B. Vasilescu, C. Casalnuovo, and P. Devanbu, "Recovering clear, natural identifiers from obfuscated JS names," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017.

[65] A.-G. Sîrbu, "Deobfuscating JavaScript code using character-based tokenization," *Studia Universitatis Babeș-Bolyai Informatica*, 2023.

[66] V. Arceri and I. Mastroeni, "Analyzing dynamic code: A sound abstract interpreter for evil eval," *ACM Transactions on Privacy and Security (ACMTOPS)*, 2021.

[67] M. AbdelKhalek and A. Shosha, "JSDES: An automated de-obfuscation system for malicious JavaScript," in *Proceedings of the 12th International Conference on Availability, Reliability and Security (ARES)*, 2017.

[68] A. Herrera, "Optimizing away JavaScript obfuscation," in *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2020.

[69] K. Hajarnis, V. Kshirsagar, S. Doiphode, and A. Joshi, "A comprehensive solution for obfuscation detection and removal based on comparative analysis of deobfuscation tools," in *Proceedings of the International Conference on Smart Generation Computing, Communication and Networking (SMART GENCON)*, 2021.

[70] C. Fang, Z. Liu, Y. Shi, J. Huang, and Q. Chen, "Large language models for code analysis: Do LLMs really do their job?" in *Proceedings of the 33rd USENIX Security Symposium (USENIX Security)*, 2024.

[71] M.-A. Lachaux, B. Roziere, M. Szafraniec, and G. Lample, "DOBF: A deobfuscation pre-training objective for programming languages," in *Proceedings of the 35th International Conference on Neural Information Processing Systems (NeurIPS)*, 2021.

[72] R. Li, C. Zhang, H. Chai, L. Ying, H. Duan, and J. Tao, "PowerPeeler: A precise and general dynamic deobfuscation method for PowerShell scripts," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024.

[73] A. Sanna, F. Cara, D. Maiorca, and G. Giacinto, "Oblivion: An open-source system for large-scale analysis of macro-based office malware," *Journal of Computer Virology and Hacking Techniques*, 2024.

[74] B. Cheng, J. Ming, J. Burket, G. Karame, and T. Dumitraş, "Obfuscation-resilient executable payload extraction from packed malware," in *Proceedings of the 30th USENIX Security Symposium*, 2021.

[75] J. Lee and W. Lee, "Simplifying mixed boolean-arithmetic obfuscation by program synthesis and term rewriting," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.

[76] B. Liu, J. Shen, J. Ming, Q. Zheng, J. Li, and D. Xu, "MBA-Blast: Unveiling and simplifying mixed Boolean-Arithmetic obfuscation," in *Proceedings of the 30th USENIX Security Symposium*, 2021.

[77] B. Reichenwallner and P. Meerwald-Stadler, "Simplification of general mixed Boolean-Arithmetic expressions: GAMBA," in *Proceedings of the 2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2023.

[78] L. Glanz, P. Müller, L. Baumgärtner, M. Reif, S. Amann, P. Anthonysamy, and M. Mezini, "Hidden in plain sight: Obfuscated strings threatening your privacy," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIA CCS)*, 2020.

[79] A. Ashraaf, H. Sarwar, G. Murtaza, and M. Ali, "DEPRODEXCATOR: A tool for decompiling and deobfuscating ProGuard and DexGuard obfuscated android applications," engrXiv preprint, 2023, accessed: Jul. 3, 2025. [Online]. Available: https://engrxiv.org/preprint/view/2910/5390

[80] P. Kochberger, S. Schrittwieser, S. Schweighofer, P. Kieseberg, and E. Weippl, "SoK: Automatic deobfuscation of virtualization-protected applications," in *Proceedings of the 16th International Conference on Availability, Reliability and Security (ARES)*, 2021.

[81] I. Khairunisa and H. Kabetta, "PHP source code protection using layout obfuscation and AES-256 encryption algorithm," in *Proceedings of the International Workshop on Big Data and Information Security (IWBIS)*, 2021.

[82] D. Zhou, L. Ying, H. Chai, and D. Wang, "From obfuscated to obvious: A comprehensive javascript deobfuscation tool for security analysis," 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2512.14070

# APPENDIX A
## OBFUSCATION CAPABILITY OF DIFFERENT TOOLS

Table IX presents a comprehensive comparison of these tools' capabilities across all 20 obfuscation techniques.

# APPENDIX B
## CFG/DDG VALIDATION EXAMPLE

Listing 3 shows a representative sample[3] that employed String Obfuscation (T3) and Eval Obfuscation (T15). CFG and DDG analysis revealed:

```javascript
var xhr = new XMLHttpRequest();
xhr.open(
  "GET", "//lurgee.download/?zBOwv=
    Q10FEhkNVgQQTxNcARAJWxRMUQ5IdRJYEhUAWB1
    ..." // truncated for brevity
);
xhr.onload = function () {
  var ref = document.referrer;
  eval(xhr.responseText);
};
xhr.send();
```

Listing 3: A representative sample before obfuscated.

- **Control Flow Analysis**: JSIMPLIFIER's deobfuscated output restored the original 8-node CFG structure with 97.30% similarity.
- **Data Dependency Verification**: DDG comparison showed 96.10% preservation rate with all variable dependencies correctly maintained.
- **Manual Verification**: Side-by-side analysis confirmed that the core functionality of downloading and executing remote JavaScript code via eval() was preserved, validating semantic consistency.

Complete CFG and DDG visualizations are available in the full version of this paper [82].

# APPENDIX C
## LLM PROMPT FOR EVALUATION

### A. Deobfuscation Prompt

> **Deobfuscation Prompt**
>
> You are a JavaScript deobfuscation expert. Please analyze the following obfuscated JavaScript code and provide the deobfuscated version. Focus on:
> 1) Restoring original variable and function names where possible
> 2) Simplifying complex expressions to their basic forms
> 3) Converting encoded strings back to readable text
> 4) Removing unnecessary complexity while preserving functionality
> 5) Ensuring the output is clean, readable JavaScript code
>
> Obfuscated code: *[OBFUSCATED_CODE]*

[3]MD5: 000c5123cd553d14cd49d22022fcc9c5

> Please provide the deobfuscated JavaScript code and what obfuscation technology the code uses.

### B. Score Readability Prompt

> **Score Readability Prompt**
>
> As a security analyst, please analyze the sample according to the following steps: 1. Identification of obfuscated techniques used for raw obfuscated samples 2. Compare and evaluate the consistency of behavior between the two samples after deobfuscation 3. The readability of both versions must be rated separately (on a 0-10 point scale), format: Obfuscated sample readability score: [X] Readability score for deobfuscated samples: [Y] Please provide the final rating directly without the need for analysis process. The rating should be based on dimensions such as code structure clarity, variable naming rationality, and logical traceability.

# APPENDIX D
## ANTI-ANALYSIS TECHNIQUE EXAMPLES

This section provides detailed code examples demonstrating how JSIMPLIFIER handles various anti-analysis techniques employed by malicious JavaScript.

```javascript
// Original obfuscated time bomb
var _0x1a2b = new Date().getTime();
var _0x3c4d = 1640995200000;
if (_0x1a2b > _0x3c4d) {
    eval(atob("dmFyIHBheWxvYWQ9Li4u"));
}

// After JSIMPLIFIER processing
var currentTimestamp = new Date().getTime();
var activationDate = 1640995200000;
if (currentTimestamp > 1640995200000) {
    eval('var payload=...');
}
```

Listing 4: Time bomb handling example (code fragment).

# APPENDIX E
## ENTROPY METRICS: FORMULATIONS AND VALIDATION

### A. Code Text Entropy Formulation

Code text entropy combines character-level and word-level frequency analysis:

$$H_{text}(X) = H_{char}(X) + H_{word}(X)$$
$$= -\sum_{i=1}^{n} p(c_i) \log_2 p(c_i) - \sum_{j=1}^{m} p(w_j) \log_2 p(w_j)$$

(1)

where $p(c_i)$ and $p(w_j)$ represent probabilities of character $c_i$ and word $w_j$ respectively.

TABLE IX: Comprehensive comparison of obfuscation capability of different tools.

| Type | Method | MeSpl | JSha | JSC | UGL | OBfu | JSV7 | DAFT | TER | FRE | STU | GNI | BEA |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Lexical | T0:Rename | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| | T1:Indirect | | ✓ | ✓ | | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ | |
| | T2:Arithmetize | ✓ | ✓ | | | ✓ | | | | ✓ | ✓ | | |
| | T3:StringEncode | ✓ | ✓ | ✓ | | ✓ | ✓ | | ✓ | ✓ | ✓ | ✓ | |
| | T4:BooleanEncode | | ✓ | ✓ | | | | | ✓ | | | | |
| Syntactic | T5:Expr2Function | | ✓ | | | | | ✓ | | | | | ✓ |
| | T6:Assign2Function | ✓ | ✓ | ✓ | | | ✓ | | | ✓ | ✓ | | |
| | T7:Reverse | | ✓ | | | ✓ | | | | | | | |
| | T8:AAEncode | | | | | | | | | | | | |
| | T9:JJEncode | | | | | | | | | | | | |
| | T10:JSFUCK | | | | | | | | | | | | |
| Semantic | T11:Arrayize | ✓ | ✓ | ✓ | | ✓ | | | ✓ | ✓ | ✓ | | |
| | T12:StrArrEncode | | ✓ | ✓ | | ✓ | | | ✓ | ✓ | ✓ | | |
| | T13:JSONEncode | | ✓ | | | | | | | | | | |
| | T14:RegexpEncode | | ✓ | ✓ | | ✓ | | | | | | | ✓ |
| | T15:Eval | | ✓ | | | | ✓ | | | ✓ | | | ✓ |
| | T16:Flattern | ✓ | ✓ | ✓ | | ✓ | | | | | | | |
| | T17:Insert | | ✓ | ✓ | | ✓ | ✓ | | | | | | |
| Multi-Lavered | T18:OB | | | | | ✓ | | | | | | | |
| | T19:LLM | ○ | ✓ | ✓ | | ✓ | ✓ | | | ✓ | ✓ | | |

Note: ✓ = fully successful, ○ = partially successful, blank = fully unsupported.

## B. AST Entropy Component Formulations

AST entropy incorporates four structural dimensions:

$$H_{node\_num}(X) = -\sum_{i=1}^{k} p(t_i) \log_2 p(t_i) \quad (2)$$

$$H_{node\_degree}(X) = -\sum_{j=1}^{l} p(d_j) \log_2 p(d_j) \quad (3)$$

where $p(t_i)$ and $p(d_j)$ represent node type and degree probabilities. Node depth and edge quantity entropies follow similar formulations.

The final AST entropy is computed as:

$$H_{AST}(X) = w_1 H_{node\_num} + w_2 H_{edge\_num} + w_3 H_{node\_degree} + w_4 H_{node\_depth} \quad (4)$$

where weights $[w_1, w_2, w_3, w_4] = [0.61, 0.79, 1.58, 1.02]$ are optimized to maximize discrimination between obfuscated and deobfuscated code through genetic algorithm.

## C. Entropy-based Evaluation Metric Validation

We validated our entropy-based approach against traditional structural similarity metrics using 60 randomly selected samples from BenignJS. Traditional methods failed on 31 out of 60 samples (51.67%), while AST entropy achieved stronger correlation with LLM-based readability assessment (r=0.53, p≤0.01) compared to structural similarity (r=0.25, p≥0.05), demonstrating superior robustness for evaluating deobfuscation effectiveness.

## APPENDIX F
### READABILITY ANALYSIS

Figure 3 presents sample-level readability improvement analysis across four LLM evaluators, demonstrating consistent effectiveness of JSIMPLIFIER in transforming obfuscated code into readable versions.
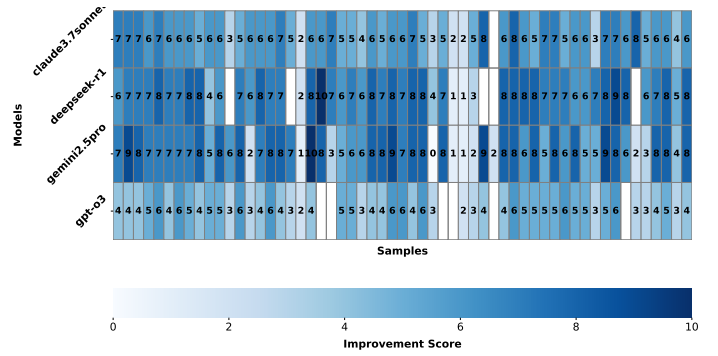


Fig. 3: Readability improvement scores validating complexity reduction across models and samples.

## APPENDIX G
### USER STUDY

The user study questionnaires are available in the full version of this paper [82]. Table X shows detailed user study results across three expertise levels.

## APPENDIX H
### FAILURE CASE EXAMPLE

This appendix presents a detailed example of semantic inconsistency introduced by LLM-based variable renaming.

### A. Original Code

```
function sumEvenSquaresOverTen(arr) {
  if (!Array.isArray(arr)) return 0;
  const filtered = arr.filter(n => n > 10 && n
    % 2 === 0);
  // ... map and reduce operations
  return sum;
}
```

Listing 5: Original clean code.

TABLE X: User study results: JSimplifier performance and relative improvements (%) compared to original and traditional conditions.

| Metric | JSimplifier Performance | | | vs. Original (%) | | | vs. Traditional (%) | | |
|---|---|---|---|---|---|---|---|---|---|
| | Novice | Intermediate | Expert | Novice | Intermediate | Expert | Novice | Intermediate | Expert |
| Average Score (100) | 53.63 | 77.34 | 75.92 | +12.7 | +5.7 | +1.5 | +5.0 | +7.0 | -4.5 |
| Average Time (s) | 549.5 | 429 | 696.5 | -16.0 | -47.7 | -2.0 | -15.8 | -47.8 | -18.6 |
| Readability (5) | 3.67 | 3.33 | 3.33 | +100 | +150 | +43 | +57 | +82 | +18 |
| Clarity (5) | 4 | 3.33 | 3.33 | +140 | +150 | +43 | +60 | +67 | +5 |
| Logicality (5) | 4 | 3 | 3.33 | +71 | +80 | +5 | +60 | +50 | +25 |
| Difficulty (5) | 3 | 2.5 | 3.17 | +100 | +25 | +9 | +50 | +36 | +12 |
| Confidence (5) | 3 | 2.5 | 4.17 | +125 | +15 | +4 | +64 | +25 | +14 |

## B. Obfuscated Code

```javascript
(function(){
 var _0xmethods = ['filter', 'map', 'reduce'];
 function _0xgetMethod(idx) {
  return _0xmethods[idx];
 }

 function sumEvenSquaresOverTen(arr) {
  if (!Array.isArray(arr)) return 0;
  var filtered = arr[_0xgetMethod(0)](function(n){
   return n > 10 && n % 2 === 0;
  });
  // ... more processing with _0xgetMethod(1), _0xgetMethod(2)
  return sum;
 }
})();
```

Listing 6: Obfuscated code with dynamic method access.

## C. Incorrectly Deobfuscated Code

```javascript
(function () {
 var arrayMethods = ["filter", "map", "reduce"];
 function getArrayMethodByIndex(methodIndex) {
  return arrayMethods[methodIndex];
 }
 function calculateSumOfEvenSquaresAboveTen(numbersArray) {
  if (!Array.isArray(numbersArray)) {
   return 0;
  }
  var evenNumbersAboveTen = numbersArray[arrayMethods[idx]](function (number) {
   return number > 10 && number % 2 === 0;
  });
  // Error: 'idx' is undefined, causing ReferenceError
  return totalOfEvenSquaredNumbers;
 }
})();
```

Listing 7: Deobfuscated code with semantic errors.

## D. Error Analysis

The deobfuscated code (Listing 7) contains several critical semantic errors compared to the original obfuscated code (Listing 6):

- **Broken Variable Dependencies:** Variable `idx` is referenced but undefined, causing `ReferenceError` at runtime.
- **Altered Control Flow:** Undefined `idx` prevents method resolution, breaking the execution flow that was correctly implemented in Listing 6.
- **Modified API Behavior:** Code attempts `arrayMethods[undefined]` instead of correct array method calls.

APPENDIX I

JAVASCRIPT OBFUSCATION TECHNIQUES CATEGORIZATION AND EXAMPLES

We evaluated 20 obfuscation techniques across 5 categories. Complete examples are available in the full version of this paper [82].

## APPENDIX J
### ARTIFACT APPENDIX

*A. Description & Requirements*

*1) How to access:* The artifact is archived on Zenodo with DOI 10.5281/zenodo.17531662 for long-term preservation and reproducibility.

The artifact is publicly available and can be accessed directly through the DOI link above. All materials, including source code, datasets, and documentation, are provided for full community transparency and reuse.

*2) Hardware dependencies:* None. JSIMPLIFIER and its evaluation experiments impose minimal hardware requirements, allowing flexible deployment on common computing platforms. Our experimental setup utilized an 8-core AMD EPYC 7742 processor with 16 GB of RAM running Windows 11; however, the tool is not restricted to specific hardware and can operate effectively on comparable or even modest systems depending on workload scale.

*3) Software dependencies:*
- Core Runtime:
    - Node.js v20.9.0
    - Python 3.11.8
- LLM Integration (Optional): API access to: OpenAI GPT, Google Gemini
- Additional Dependencies:
    - All Python packages listed in `requirements.txt`
    - All Node.js packages listed in `package.json`

*4) Benchmarks:*
- Primary Datasets:
    - MalJS: 23,212 real-world obfuscated malicious JavaScript samples (391.78 KB average size)
    - BenignJS: 21,209 real-world obfuscated benign JavaScript samples (41.40 KB average size)

*B. Artifact Installation & Configuration*

1) **Download and extract the artifact:**
   ```
   # Download from Zenodo
   # Extract the downloaded archive
   unzip jsimplifier.zip
   cd jsimplifier
   ```

2) **Install Node.js dependencies:**
   ```
   npm install
   ```

3) **Install Python dependencies:**
   ```
   pip install -r requirements.txt
   ```

4) **Configure LLM API access (optional):**
   Set environment variables for API access:
   ```
   export OPENAI_API_KEY="your-openai-key"
   export GEMINI_API_KEY="your-gemini-key"
   export BASE_URL="your-baseurl"
   ```

   Or provide API keys directly via command line parameters during execution.

*C. Experiment Workflow*

The experimental evaluation is designed for practical artifact assessment with flexible scaling options to accommodate different evaluation time constraints:

**Stage 1**: Deobfuscation Capability Demonstration - Interactive testing of individual obfuscation techniques with provided sample cases

**Stage 2**: Syntax Validation - Automated verification that deobfuscated output is syntactically correct JavaScript

**Stage 3**: Scalable Effectiveness Validation - Process real-world samples with configurable dataset size (from quick demos to full-scale evaluation)

Each stage provides clear, verifiable outputs with time estimates ranging from minutes (demo mode) to days (full evaluation).

*D. Major Claims*

- **(C1)**: JSIMPLIFIER successfully handles all 20 categorized obfuscation techniques. This is demonstrated by experiment (E1) using provided sample cases for each technique.
- **(C2)**: JSIMPLIFIER produces syntactically valid JavaScript output. This is validated by experiment (E2) using automated syntax checking.
- **(C3)**: JSIMPLIFIER demonstrates robust processing capability on real-world samples at scale. This is shown by experiment (E3) with configurable dataset sizes to balance evaluation thoroughness with time constraints.

*E. Evaluation*

*1) Experiment (E1): Deobfuscation Capability Demonstration:* **[15 human-minutes + 10 compute-minutes]**: Validate JSIMPLIFIER's capability to handle different obfuscation techniques using provided sample cases.
*[Preparation]*

```
cd experiments/E1
```

*[Execution]*

```
# Test individual techniques
npm start deobfuscate -- --model=... \
  --apiKey=... --baseURL=... \
  experiments/E1/obfuscated_samples/T0_Rename.js
npm start deobfuscate -- --model=none \
  experiments/E1/obfuscated_samples/T1_Indirect.js
# ... or run all samples
python .\run_technique_demos.py --model ... \
  --api-key ... --base-url ...
```

*[Results]*
Expected: Each sample produces readable, deobfuscated output. Results saved in `results/` with side-by-side comparisons.

*2) Experiment (E2): Syntax Validation:* **[10 human-minutes + 5 compute-minutes]**: Verify that deobfuscated output is syntactically valid JavaScript.
*[Preparation]*

```
cd experiments/E2
```

*[Execution]*

```
# Validate syntax of deobfuscated files
python validate_syntax.py \
  ../E1/results/
```

*[Results]*

Expected: 100% syntax validation success. Output shows parsing results for each deobfuscated file.

*3) Experiment (E3): Scalable Effectiveness Validation:*
**Time Options: [Quick: 30 minutes] [Standard: 4 hours] [Full: 8 days single-core, 1-2 days multi-core]**: Demonstrate tool robustness on real-world malicious JavaScript samples with flexible scale and parallelization options.

*[Preparation]*

```
tar -zxvf dataset/MalJS.tar.gz
cd experiments/E3
```

*[Execution Options]*
**Quick Demo (30 minutes):**

```
# Process 50 representative samples
python run_validation.py \
  --dataset quick_demo \
  --workers 1
```

**Standard Evaluation (4 hours):**

```
# Process 500 samples with multi-core
python run_validation.py \
  --dataset standard_eval \
  --workers 4
```

**Full Dataset Processing:**

```
# Single-core (~8 days):
python run_validation.py \
  --dataset full_dataset \
  --workers 1

# Multi-core (1-2 days):
python run_validation.py \
  --dataset full_dataset \
  --workers 8
# Adjust --workers based on
# available cores
```

*[Results]*

Expected: Entropy reduction validation demonstrating deobfuscation effectiveness. The script generates box plots comparing entropy values before and after deobfuscation, showing significant complexity reduction across processed samples. See `output_validation_quick_demo\entropy_comparison_boxplot.png` in the root directory for visual confirmation of entropy reduction results

*F. Notes*

**Dataset and Cost Considerations:** The complete MalJS dataset (23,212 samples) is provided for full reproducibility. LLM-based experiments require API access and incur approximately $480-500 for full evaluation using GPT-4o-mini, though all core claims can be validated without LLM processing using AST-only mode. Scaled subsets maintain statistical representativeness while reducing both evaluation time and costs.

**Performance Characteristics:** JSIMPLIFIER processes samples at an average rate of 28 seconds per file (excluding LLM processing), with memory requirements scaling linearly with dataset size (8GB sufficient for standard evaluation). Full dataset processing requires approximately 8 days on single-core systems or 1-2 days with 8+ cores, as performance is primarily CPU-bound. Network connectivity is only required for optional LLM-based enhancements.

**Evaluation Flexibility:** The artifact supports three evaluation scales: quick validation (30 minutes) verifies core functionality with representative samples, standard evaluation (4 hours) provides statistical confidence through multi-core processing, and full reproduction (1-3 days depending on available cores) processes the complete dataset. LLM processing adds significant overhead and is not recommended for full dataset evaluation due to cost and time constraints.