JsShaker: Dependency-Tracked Abstract Interpretation for JavaScript Code Size Optimization

TONGRUI XIONG, University of Science and Technology of China, China
BOYAO DING, University of Science and Technology of China, China
YU ZHANG, University of Science and Technology of China, China
YIFAN XU, University of Science and Technology of China, China
MINGKAI XU, University of Science and Technology of China, China

JavaScript code size is a critical factor in Web performance. Existing code size optimization techniques rely primarily on peephole optimizations and lack whole-program inter-procedural static analysis, leaving them severely constrained by JavaScript's dynamic nature. While abstract interpretation offers the necessary semantic insight to overcome this, it typically requires expensive, iterative analysis–transformation cycles, rendering it impractical for production pipelines.

To bridge this gap, we present JsShaker, a framework leveraging dependency-tracked abstract interpretation. By introducing dependency-aware abstract values, JsShaker determines the necessity of code constructs simultaneously with control and data flow analysis, enabling an efficient single-round analysis-then-transform approach. Beyond foundational tree-shaking, the framework is extensible for advanced optimizations. We demonstrate this by modularly integrating branch folding, constant folding, and the first effective safe property mangling. We evaluated JsShaker on 13 target programs composed of widely deployed, real-world JavaScript libraries. Results show that JsShaker yields an average additional code size reduction of 27.6% (peaking at 66.7%) while incurring only a 14% build-time overhead over the baseline pipeline, outperforming state-of-the-art tools (e.g., Google Closure Compiler v20260128). Furthermore, JsShaker passes all relevant tests in the official Test262 conformance suite, ensuring correctness.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Automated static analysis**.

Additional Key Words and Phrases: abstract interpretation, code size optimization, tree-shaking, JavaScript

1 Introduction

Originally designed for browsers, JavaScript has evolved into one of the world's most widely used programming languages [GitHub 2025]. Beyond dominating the modern Web, it now powers diverse computing environments, spanning server-side platforms [Node.js 2009], desktop applications [Electron 2013], mobile apps [React Native 2015], and even IoT [JerryScript 2015].

To facilitate code reuse, modern JavaScript applications rely heavily on the npm ecosystem [npm, Inc. 2026], importing an average of about 350 packages [Sonatype 2024, Fig. 4.4, p. 52]. However, because these applications typically invoke only a subset of the functionality provided by their dependencies, a notable portion of the imported code remains effectively unused at runtime. This unutilized code, combined with standard developer practices—such as extracting constants, writing development-specific logic, and using descriptive identifiers—collectively leads to code bloat.

This code bloat severely impacts Web performance and business metrics. Large bundles delay initial content display due to longer network downloads and costly parsing and compilation overhead on the CPU; in contrast, even minor speed improvements can significantly boost user experience and revenue [Chaqfeh et al. 2022b; Chrome Team 2023; Google 2019]. Furthermore, emerging deployment paradigms impose strict size constraints. Serverless functions deployed at the network edge require fast startup times, tightly limiting script sizes (e.g., Cloudflare Workers'

Authors' Contact Information: Tongrui Xiong, xtr@mail.ustc.edu.cn, University of Science and Technology of China, China; Boyao Ding, via@mail.ustc.edu.cn, University of Science and Technology of China, China; Yu Zhang, yuzhang@ustc.edu.cn, University of Science and Technology of China, China; Yifan Xu, zjxyfan@mail.ustc.edu.cn, University of Science and Technology of China, China; Mingkai Xu, xumk@mail.ustc.edu.cn, University of Science and Technology of China, China.

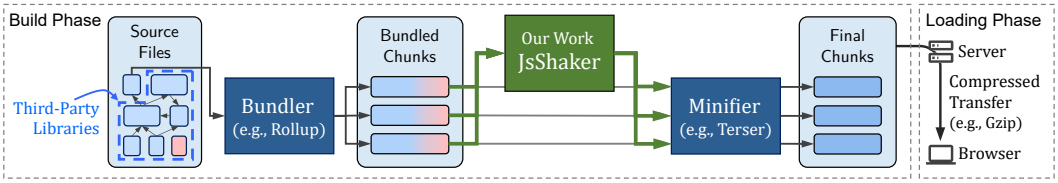

Fig. 1. A typical JavaScript code size optimization pipeline.

3 MB size limit [Cloudflare 2025]). Similarly, JavaScript's growing adoption in memory-constrained embedded devices demands a minimal code size [Ugawa et al. 2019].

To mitigate code bloat, a two-step optimization process is widely adopted (Fig. 1). First, bundlers prune unused module-level items via reachability analysis, where Rollup [Harris et al. 2026] stands as the pioneer and newer tools like Rolldown [VoidZero Inc. 2026] significantly accelerate the process. Second, minifiers generate syntactically compact code; while Terser [Maiwald et al. 2026] remains the standard, faster alternatives such as swc [Kang et al. 2019], esbuild [Wallace 2020], and oxc-minify [Chen et al. 2026] are increasingly used. Despite their ubiquity, these two steps fundamentally lack the semantic insight required to identify data-dependent dead code. Moreover, advanced whole-program optimizers like Google Closure Compiler (Advanced mode) [Google 2026] attempt deeper reductions through aggressive inlining and unsafe property renaming. However, lacking precise heap modeling, they cannot safely specialize dynamically dispatched functions or optimize computed property accesses. Recent research has explored further optimizations: Lacuna [Malavolta et al. 2023] constructs call graphs to remove unreachable functions, while DepPrune [Liu et al. 2025] identifies unused CommonJS packages via system call monitoring. Yet neither work supports expression-level optimizations. Broader efforts to improve load times include script filtering [Chaqfeh et al. 2022b, 2020] and the invention of minimalist browser-native languages [Pandey et al. 2025]. Despite these efforts, the fundamental gap remains: the absence of a safe, high-performance, and fine-grained optimization tool built on precise semantic understanding.

Abstract interpretation provides a framework for such semantic analysis, evidenced by tools like JSAI [Kashyap et al. 2014], SAFE [Lee et al. 2012; Park et al. 2017b], and FAST [Kang et al. 2023] that precisely model highly dynamic JavaScript features. However, applying high-precision abstract interpretation to code optimization faces a major hurdle: the *circular dependency* between analysis and transformation. Because transformations alter program structure and unlock new optimization opportunities, compilers are traditionally forced into an expensive, multi-pass analysis–transformation cycle. Although theoretically possible, unifying these phases into a single fixpoint computation is prohibitively expensive and intractable given JavaScript's dynamic semantics (Section 2.2). The JavaScript runtime performance optimizer Prepack [Facebook, Inc. 2017] avoided this cycle via symbolic execution and heap serialization, but suffered from excessive complexity and struggled to support full JavaScript semantics, leading to its discontinuation [Herman 2020].

In this paper, we present JsShaker, a novel code size optimizer that reconciles high-precision analysis with build-time efficiency. While leveraging abstract interpretation to capture deep semantics, JsShaker introduces a key innovation: *dependency-aware abstract values*, which augment abstract values with dependency information to lift tree-shaking to a fine-grained semantic level. This enables essential code constructs to be identified during data-flow analysis and supports a single-round analysis–transformation workflow, avoiding costly iterative cycles. Based on this framework, we also implement modular optimizations beyond foundational tree-shaking, most notably the first effective realization of safe property mangling.

The key contributions of this paper are as follows:

(1) We propose JsShaker, a JavaScript code-size optimization framework that achieves fine-grained tree-shaking. By leveraging dependency-tracked abstract interpretation, it determines code liveness as an integrated part of value inference to guide the subsequent code transformation, eliminating the need for iterative analysis–transformation cycles (Section 3).
(2) We design a context-/object-/field-sensitive abstract interpretation method tailored for JavaScript's dynamic features. This method precisely models complex control flows and object semantics, and incorporates function summaries, striking a practical balance between analysis precision and build-time efficiency (Section 4).
(3) We demonstrate the framework's extensibility by modularly implementing advanced optimizations atop foundational tree-shaking, including branch folding, constant folding, and the first effective realization of safe property mangling (Section 5).
(4) We evaluate JsShaker on 13 programs involving real-world JavaScript libraries. The results show that JsShaker yields an average additional code size reduction of 27.6% (peaking at 66.7%) while incurring only a 14% execution time increase over the baseline pipeline. Moreover, it passes all relevant test cases in the Test262 conformance suite [Ecma TC39 2026], ensuring strict correctness and compatibility with modern language standards (Section 6).

2 Background and Motivation

2.1 Limitations of Existing JavaScript Code Size Optimizers

Over the past two decades, numerous tools have been developed to mitigate JavaScript code bloat. While diverse in implementation, they generally fall into four primary categories based on their underlying analysis techniques, as summarized in Table 1. However, a fundamental trade-off between optimization effectiveness and efficiency remains unresolved. Reachability analysis and multi-pass AST rewriting achieve build-time scalability but lack the semantic insight to resolve complex data-dependent dead code. Conversely, coverage-guided pruning can achieve deeper optimizations, but incurs prohibitive computational overhead and risks unsafe code elimination. Abstract interpretation, while theoretically promising, has yet to be successfully applied in practice. Script filtering techniques are not applicable to bundled applications.

Reachability Analysis. These techniques identify reachable code from entry points using module-level import graphs (e.g., Rollup) or function-level call graphs (e.g., Lacuna). However, they fail to eliminate code that is structurally reachable but functionally unused at runtime.

Multi-Pass AST Rewriting. These tools iteratively transform the AST for optimizations like syntactic rewriting (e.g., white space removal), function inlining, and local dead code elimination. However, their reliance on predefined patterns lacks the semantic insight required to identify data-dependent dead code, restricting them to source code improvement [Farzat et al. 2021].

Table 1. JavaScript code size optimizers

Technique	Tools
Reachability Analysis	Rollup [Harris et al. 2026], Webpack [Koppers et al. 2017], Mininode [Koishybayev and Kapravelos 2020], Rolldown [VoidZero Inc. 2026], Lacuna [Malavolta et al. 2023]
Multi-Pass AST Rewriting	Google Closure Compiler [Google 2026], UglifyJS [Bazon 2010], Terser [Maiwald et al. 2026], swc [Kang et al. 2019], esbuild [Wallace 2020], oxc-minify [Chen et al. 2026]
Abstract Interpretation	Prepack *(Runtime Performance Optimizer)* [Facebook, Inc. 2017]
Workload-Guided Pruning	DFAHC [Farzat et al. 2021], Muzeel [Kupoluyi et al. 2022], JSAnalyzer [Chaqfeh et al. 2022a], Stubbifier [Turcotte et al. 2022], DepPrune [Liu et al. 2025]
Script Filtering	JSCleaner [Chaqfeh et al. 2020], SlimWeb [Chaqfeh et al. 2022b]

Abstract Interpretation. Though designed for execution performance, Prepack demonstrated abstract interpretation's potential for code size reduction through constant folding and function specialization. However, its rigid symbolic execution and inherently intricate heap serialization imposed excessive complexity and usage restrictions, leading to its discontinuation [Herman 2020].

Workload-Guided Pruning. These techniques dynamically execute code—often via test suites or automated interactions—to observe and prune unused paths. However, this approach is fundamentally unsafe, as incomplete coverage inevitably causes the erroneous removal of critical edge-case code. Furthermore, multiple execution rounds make the process inherently slow.

Script Filtering. Operating at the network or browser layer, these techniques entirely block scripts deemed non-critical. However, this coarse, file-level granularity makes them incompatible with modern bundled applications, where critical and non-critical code coexist within the same file.

2.2 Abstract Interpretation with Tree-Shaking

Abstract interpretation provides a formal framework for safely approximating a program's runtime behavior [Cousot and Cousot 1977], which can be used to identify dead code with desired precision. However, the traditional interdependence between analysis and transformation forces multiple rounds of expensive abstract interpretation. For example, fully optimizing Listing 1 requires three rounds: first, constant folding evaluates `f("42")` to `42`; second, branch folding eliminates the `if` structure; finally, dead code elimination removes the unused `null` argument.

```
1 function f(x) { if(x) return +x; else return input(); }
2 console.log(f("42"), f(null));
```

```
1 function f() { return input(); }
2 console.log(42, f());
```

Listing 1. An example requiring multiple analysis–transformation cycles for full optimization.

While intertwining abstract interpretation with code optimization is possible [Lerner et al. 2002; Lesbre and Lemerre 2024], applying such techniques to JavaScript is exceptionally complex: the language's dynamic features render maintaining an efficiently updatable call graph or data flow graph intractable, and its polymorphic nature makes context-insensitive approaches imprecise.

Tree-shaking is an algorithmic approach that complements abstract interpretation in resolving this dependency cycle. Originating in LISP and adopted across various object-oriented languages [Condit et al. 2008; Grove et al. 1997], tree-shaking has seen a widespread resurgence in JavaScript. Although often narrowly understood as module-level pruning in bundlers, it fundamentally represents an algorithmic philosophy distinct from classical Dead Code Elimination (DCE) [Harris 2015]. While DCE operates on an *exclusionary* basis—identifying and removing provably inactive instructions—tree-shaking uses an *inclusionary* approach, tracing from entry points to retain only live code. Proving a code fragment "dead" (DCE) requires verifying the absence of side effects across all data paths, which is often undecidable due to dynamic features and complex aliasing. Conversely, tracing liveness flows (tree-shaking) offers a feasible and safe approximation. By integrating tree-shaking into abstract interpretation, it is possible to achieve deep optimization in a single analysis–transformation round.

2.3 Challenges

To effectively optimize code size, we base our approach on abstract interpretation with tree-shaking. We identify the primary challenges as follows:

2.3.1 Handling Control Flow Ambiguity. JavaScript treats functions as first-class citizens and relies heavily on closures (e.g., line 2 of Listing 2). This dynamic nature obscures call targets, making it

```
const obj = {};
document.onclick = function() { console.log(obj.foo); }; // escaped callback function
obj[ThirdPartyLib.check() ? 'foo' : 'bar'] = 2; // ThirdPartyLib is opaque to the analysis
Object.setPrototypeOf(obj, { foo: 3 }); // dynamic prototype modification: obj.foo is now 2 or 3
```

Listing 2. An illustrative example demonstrating how JavaScript's dynamic features complicate static analysis.

difficult to construct a precise static control-flow graph for interprocedural analysis. Furthermore, the prevalence of asynchronous and generator functions renders execution timing highly unpredictable [Lucas et al. 2025]. To achieve safe optimization, the analysis must conservatively consider all possible call targets and execution interleavings.

2.3.2 Resolving Dynamic Object Fields. JavaScript's prototype-based paradigm allows object structures to be dynamically mutated. As demonstrated in Listing 2, properties can be dynamically created or accessed via computed names, and prototype chains can be arbitrarily altered. This poses a fundamental challenge for static analysis, especially field-sensitive analysis [Sun and Ryu 2018].

2.3.3 Tolerating Missing Definitions in Partial Programs. JavaScript applications interact with diverse host environments and dynamically loaded third-party scripts. For example, `ThirdPartyLib.check()` in Listing 2 represents a call to an external API with an opaque implementation. Unlike most compiled languages where external interfaces are strictly bound, exhaustively pre-declaring all external interactions is infeasible in JavaScript. Therefore, optimizers must treat such codebases as partial programs, safely over-approximating unknown behaviors to preserve soundness while offering extensibility for precise environmental modeling.

2.3.4 Constraints of Source-to-Source Transformation. Traditional compilers typically lower the AST into an Intermediate Representation (IR), such as static single assignment, to simplify analysis and transformations. However, reconstructing high-level JavaScript from a low-level IR yields verbose code requiring complex semantic recovery (e.g., re-synthesizing a compact `while (t) { ... }` loop from a flattened, `switch`-based IR). This inherent bloat defeats the primary goal of source-to-source code size reduction. Consequently, the optimizer must perform semantic analysis directly on the JavaScript AST, significantly complicating the analysis engine.

2.3.5 Strict Build-Time Performance Requirements. As modern JavaScript codebases expand, prolonged build times have emerged as a critical developer pain point [Devographics 2025]. Conventional optimizers routinely employ multi-round analysis–transformation cycles. While this iterative approach is viable for computationally cheap syntactic analyses, abstract interpretation is inherently much more expensive. As a result, applying such deep semantic analysis iteratively becomes computationally prohibitive for production pipelines (Section 2.2). To meet strict build-time constraints, a practical semantic optimizer must eliminate these multi-round loops and ensure the abstract interpretation itself is highly efficient.

3 Design Overview

This section outlines the design goals of JsShaker (Section 3.1), followed by the system workflow (Section 3.2), the core dependency tracking mechanism (Section 3.3), and the code transformations it drives (Section 3.4).

3.1 Design Goal

The primary objective of JsShaker is to minimize JavaScript code size by identifying and eliminating redundant computations via deep semantic analysis. To achieve this in a production-grade build tool, we adhere to four key design goals:

G1 *High Precision.* The analysis must go beyond simple reachability. JsShaker incorporates context-/object-/field-sensitivity to precisely model the heap and value flow, maximizing the identification of dead code hidden within complex logic.

G2 *Efficiency.* To address the precision-efficiency gap inherent in semantic optimization, the system must avoid the prohibitive cost of iterative analysis–transformation cycles. We aim to achieve aggressive optimization in a single round through our novel dependency-aware abstract value.

G3 *Soundiness.* Full soundness is often impractical for JavaScript due to features like `eval`. Instead, we aim for *soundiness* [Livshits et al. 2015] by making best-effort but unsafe approximations for a minimal set of analysis-defeating features, while maintaining safety for the rest.

G4 *Architectural Adaptability.* Given the diverse runtime environments of JavaScript and the domain-specific semantics of modern ecosystems, the system must be extensible for custom host environments and new optimization capabilities without fundamental modification.

3.2 System Workflow

Fig. 2 illustrates the overall design of JsShaker, consisting of two parts:

Single-Round Analysis and Transformation Workflow. The core of JsShaker is a streamlined two-phase workflow: *Analysis* and *Transformation*. Crucially, to meet the performance requirements, each phase is performed only once; the system does not re-run analysis after transformation. Formalized in Section 4, the analysis phase abstractly interprets the input AST to overcome the limitations of IR-based approaches (Section 2.3.4). Unlike traditional abstract interpreters that operate on abstract values alone, our analysis operates on *DepValues*—abstract values augmented with dependency information. As the interpreter traverses the code, it dynamically identifies essential constructs and optimization opportunities, accumulating them into the global liveness set $\mathcal{L}$. Detailed in Section 3.4, the transformation phase then reconstructs the AST. By querying $\mathcal{L}$, it determines which code constructs to retain, optimize, or eliminate, effectively performing tree-shaking and advanced optimizations in a single pass.

Extensible and Modular Optimizations. While the core workflow is capable of performing effective tree-shaking, it also serves as a foundation for advanced optimizations. In Section 5, we illustrate this extensibility via branch folding, constant folding, and property mangling, which further reduce code size by modularly integrating into the workflow through the incorporation of new dependency variants and transformation logic.

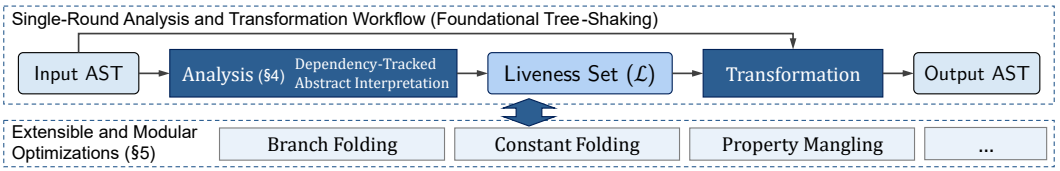

Fig. 2. The overall design of JsShaker. Beyond the core single-round workflow (upper part), the system also supports modular and extensible optimizations (lower part).

3.3 Dependency-Aware Abstract Value

As the key innovation, dependency-aware abstract values (*DepValues*) enable the analysis to identify essential code constructs and optimization opportunities in a single run. Next, we first define the abstract domain and *DepValues*, then explain how dependencies are propagated and included.

3.3.1 Abstract Domain. A *DepValue* $e = \langle v, d \rangle$ (Fig. 5) consists of an abstract value v and a dependency set d. The value component v is defined over an abstract domain $\mathcal{V}$ that approximates the concrete execution behaviors of JavaScript [Cousot and Cousot 1977]. Fig. 3 illustrates the lattice of this domain $\mathcal{V} = \{\top, \bot\} \cup \mathcal{V}_\mathrm{P} \cup \mathcal{V}_\mathrm{R} \cup \mathcal{V}_\mathrm{F} \cup \mathcal{V}_\mathrm{E} \cup \mathcal{V}_\mathrm{U}$, comprising seven components:

- *Top* $\top$. The top element of the lattice, representing any possible JavaScript value.
- *Bottom* $\bot$. The bottom element of the lattice, representing unreachable code paths.
- *Primitives* $\mathcal{V}_\mathrm{P}$. In JavaScript, a primitive is data that is not an object and has no instance properties. There are 7 types of primitives: string, number, bigint, boolean, undefined, symbol, null [MDN Contributors 2025]. We model them at two levels of precision: (1) literal values, such as the number 42; (2) type-bounded primitives, such as an unknown boolean $\top_\mathrm{Bool}$.
- *Records* $\mathcal{V}_\mathrm{R}$. As formalized in Fig. 4, for ordinary objects, we track the prototype v_proto and maintain a property map μ for field-sensitive analysis. This map associates abstract keys k with their existence status ξ and possible descriptors Θ. Abstract keys include string literals (known keys), Rest (other keys), and $\top$ (any keys). A descriptor θ is either a data descriptor (containing a memory address a; see Definition 4.2) or an accessor descriptor (containing a getter e_get and a setter e_set).
- *Functions* $\mathcal{V}_\mathrm{F}$. JavaScript functions are first-class objects. We only model functions with known behavior, including user-defined functions (i.e., closures) f and built-in functions. The top element of this domain is $\top_\mathrm{F}$. The modeling of user-defined functions is formalized in Section 4.3.1.
- *Exotic Objects* $\mathcal{V}_\mathrm{E}$. As defined in the ECMAScript specification [Ecma International 2025], exotic objects (e.g., arrays) exhibit behaviors distinct from ordinary objects. We design specialized abstract values for these exotic objects to accurately capture their unique behaviors during analysis (formalization omitted for brevity).
- *Union* $\mathcal{V}_\mathrm{U}$. Instead of resolving the join of distinct abstract types to $\top$, which is common in JavaScript due to dynamic typing, we represent the join precisely as a union of the original abstract values. Fig. 3 illustrates a union of 42 and unknown string.

3.3.2 Dependencies. The second component d of a *DepValue* e (Fig. 5) is a dependency set capturing *why* the value exists, i.e., which code constructs and program semantics contribute to its creation and flow. While extensible for further optimizations, the core variant of a dependency δ is the

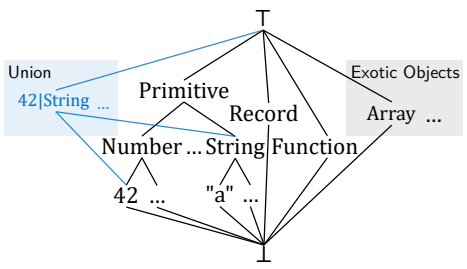

Fig. 3. Overall structure of the abstract domain.

$$\mathcal{V}_\mathrm{R} \ni \mathrm{Record}(v_\mathrm{proto}, \mu)$$

$$\begin{aligned}
(\textit{PropMap})\ \ \mu &::= \{k \mapsto \langle \xi, \Theta \rangle\} \\
(\textit{PropKey})\ \ k &::= s \mid \mathrm{Rest} \mid \top \\
(\textit{Existence})\ \ \xi &::= \mathrm{Definite} \mid \mathrm{Maybe} \\
(\textit{Descriptors})\ \Theta &::= \{\theta\} \\
(\textit{Descriptor})\ \ \theta &::= \mathrm{Data}(a) \mid \mathrm{Accessor}(e_\mathrm{get}, e_\mathrm{set}) \\
(\textit{Address})\ \ a &\ \text{(defined in Definition 4.2)}
\end{aligned}$$

Fig. 4. The definition of abstract ordinary objects.

$$
\begin{aligned}
\text{(DepValue)}\quad & e ::= \langle v, d \rangle \\
\text{(AbstractValue)}\quad & v \in \mathcal{V} \\
\text{(DependencySet)}\quad & d ::= \{\delta\} \\
\text{(LivenessSet)}\quad & \mathcal{L} ::= \{\delta\} \\
\text{(Dependency)}\quad & \delta ::= \ell \mid \cdots \text{(extensible)} \\
\text{(CodeLocation)}\quad & \ell ::= i \text{ (unique identifier)}
\end{aligned}
$$

Fig. 5. Formal definition of dependency-aware abstract values.

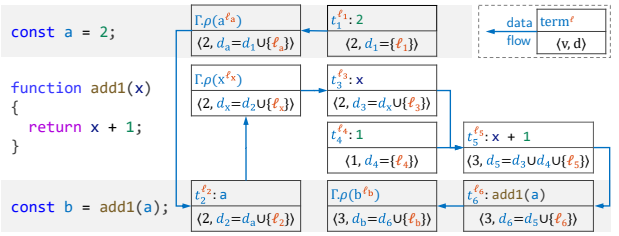

Fig. 6. An example of data flow with dependency-aware abstract values.

Code Location ℓ, which uniquely identifies each syntactic construct in the source code (typically implemented as an AST node pointer). The notation t^{ℓ} indicates that term t occurs at location ℓ.

We define the join operation for two *DepValues* as $\langle v_1, d_1 \rangle \sqcup \langle v_2, d_2 \rangle \triangleq \langle v_1 \sqcup v_2, d_1 \cup d_2 \rangle$, where abstract values undergo a standard lattice join while dependency sets are merged via set union to ensure that all contributing dependencies are preserved during analysis. Since our algorithm avoids element-wise lookups within the dependency sets, we can efficiently implement them as concatenation trees, achieving $O(1)$ non-destructive merging by simply creating new root nodes to link immutable subtrees. Although our implementation does not explicitly free dependency sets, it scales to industrial code without memory exhaustion as evaluated in Section 6.2.

3.3.3 Dependency Propagation Mechanism. Fig. 6 illustrates the core mechanics of interprocedural dependency propagation through a simple program. As data flows through the AST, each evaluated syntactic construct appends its code location ℓ_i to the dependency set of the resulting *DepValue*. For example, evaluating the literal `1` generates a base *DepValue* $\langle 1, \{\ell_4\} \rangle$. When a computation derives from multiple values, such as the addition `x + 1` at ℓ_5, the analysis computes the abstract value and unions the underlying dependencies, yielding $\langle 3, d_3 \cup d_4 \cup \{\ell_5\} \rangle$. This propagation naturally crosses function boundaries: the dependencies of argument `a` (d_2) flow into parameter `x`, propagate through function `add1`, and return to variable `b`. Consequently, the final *DepValue* encapsulates the precise provenance of all contributing code constructs.

3.3.4 Liveness Set and Dependency Inclusion. The core output of the analysis phase is the *Liveness Set* $\mathcal{L}$, a global set of dependencies deemed essential for the program's observable behavior. During abstract interpretation, dependencies are latent until they flow into an operation that produces an observable side effect or escapes the analysis scope. Formally, at evaluation steps invoking the SideEffect or Escaped helpers (both defined in Section 4), all relevant dependencies are added to the liveness set $\mathcal{L}$. This inclusion process is strictly monotonic: once a dependency is included in $\mathcal{L}$, it is permanently marked as "live," ensuring the preservation of its semantics.

3.4 Code Transformation

The final phase of JsShaker reconstructs the input AST into an optimized form, guided by the *Liveness Set* $\mathcal{L}$ computed during the analysis phase. This process is formalized as a syntax-directed transformation function, denoted as $\widehat{t}$, which recursively processes statements and expressions. The following is the key transformation logic that achieves tree-shaking, while extended advanced optimizations demonstrated in Section 5 are defined atop these foundational rules:

- For a statement t, the transformation is determined by its control-flow reachability. If t is reached during abstract interpretation, $\widehat{t}$ preserves its original syntactic structure while recursively transforming all sub-terms. Conversely, unreachable statements are eliminated.

- For an expression t at location ℓ, the transformation relies on the *Liveness Set* $\mathcal{L}$. If the expression is live ($\ell \in \mathcal{L}$), $\widehat{t}$ retains its structure and recursively transforms its operands. Otherwise, it is reduced to a valid and minimal code construct composed of the non-empty transformed sub-terms, ensuring essential side effects are preserved while the unused result is discarded. For instance, $(t_1[t_2])^\ell$ would be transformed to $\widehat{t_1}[\widehat{t_2}]$ if $\ell \in \mathcal{L}$ holds, and to $(\widehat{t_1}, \widehat{t_2})$ (a comma expression, which can be further reduced if any of the sub-terms is empty) otherwise.

4 Abstract Interpretation

In this section, we present the abstract interpretation framework of JsShaker. We first describe the control flow analysis (Section 4.1), which provides the execution scaffolding. Next, we detail the data flow analysis (Section 4.2), focusing on how it addresses JavaScript's dynamic object semantics while incorporating the dependency tracking mechanism. Finally, we present a context-sensitive interprocedural analysis, augmented by function summaries for improved efficiency (Section 4.3).

For the sake of clarity and brevity, we restrict our formalism to a core subset of JavaScript, omitting features such as exceptions, classes, `this`, and rest parameters. However, our implementation fully supports the ECMAScript 2025 specification [Ecma International 2025], verified in Section 6.3.

To avoid the overhead of maintaining and merging multiple dependency-aware environments, our abstract interpretation sacrifices path-sensitivity and uses a single global analysis environment Γ for the overall state.

Definition 4.1 (Global Analysis Environment Γ). $\Gamma = \langle \psi, \rho \rangle$, where $\Gamma.\psi$ represents the control environment (see Definition 4.5) and $\Gamma.\rho$ represents the lexical environment $\rho : \mathit{Identifier} \rightarrow \mathit{Address}$, which maps identifiers to memory addresses and models variable scoping and mutable state.

Definition 4.2 (Memory Address a). A *Memory Address* $a = \langle e, \psi \rangle \in \mathit{Address}$ is a storage location holding a *DepValue* e and the control environment ψ captured at allocation. Updates to a modify e while preserving ψ, enabling precise tracking of determinism and control dependencies.

4.1 Control Flow Analysis

Control flow analysis provides the execution scaffolding for our abstract interpretation framework. To soundly capture JavaScript's dynamic semantics, the analysis must maintain precise context awareness and strictly coordinate control-state transitions with data-flow updates.

4.1.1 Control State, Frame and Stack. A *Control State* characterizes the reachability and determinism of a *Control Frame*, which forms the global *Control Stack* modeling the nested control structures. Execution proceeds either by normal sequential execution or by abrupt transfer, which redirects control to a target. We distinguish three control states as follows:

Definition 4.3 (Control State σ). A *Control State* is $\sigma ::= \mathsf{Normal} \mid \mathsf{Unsure} \mid \mathsf{Abrupt}$, where:

- Normal represents a deterministic sequential path.
- Unsure represents non-deterministic reachability (e.g., due to unknown branch conditions).
- Abrupt indicates that the current execution point is unreachable due to control-flow abruption.

Reachable control states are denoted as $\sigma^+ \in \{\mathsf{Normal}, \mathsf{Unsure}\}$. To handle nested control structures, we define a composition operator $\sigma^+ \circ \sigma \triangleq (\sigma^+ = \mathsf{Unsure}\ \mathbf{?}\ \mathsf{Unsure}\ \mathbf{:}\ \sigma)$, which captures the propagation of non-determinism.

Definition 4.4 (Control Frame ψ). A *Control Frame* $\psi ::= \langle \kappa, \sigma, d \rangle$ captures the context of a scoped control structure, where κ identifies the abrupt target kind (e.g., $\mathsf{FuncCall}(f)$, BreakTarget, ContinueTarget), σ tracks the dynamic control state within this scope, and d accumulates the control dependencies detailed in Section 4.1.2.

Definition 4.5 (Control Stack). Environment Γ maintains control frames as a stack, with $\Gamma.\psi$ denoting the top frame. Entering a control structure pushes a frame $\langle \kappa, \sigma, d \rangle$ via $\Gamma.\text{Push}(\kappa, \sigma, d)$; exiting pops it via $\Gamma.\text{Pop}()$.

4.1.2 *Determinism and Dependency Propagation.*

Sequential Execution. By default, statements are evaluated sequentially until the active control state $\Gamma.\psi.\sigma$ becomes Abrupt, at which point execution exits the current block.

Control Flow Unwinding. Such control state transitions are triggered by abrupt constructs (`break`, `continue`, `return`). For example, evaluating break^{ℓ} calls $\Gamma.\text{UnwindTo}(\text{BreakTarget}, \{\ell\})$ (defined in Algorithm 1). In Fig. 7, the `break` at ① first updates ψ_2 to Abrupt (stopping the inner flow), then updates the parent ψ_1 with the compounded state Unsure ∘ Abrupt = Unsure (②), correctly marking subsequent statements as non-deterministic instead of unreachable.

Control Dependencies. Unlike traditional pre-constructed post-dominator trees [Ferrante et al. 1987], JsShaker captures control dependencies on-the-fly via $\psi.d$. Entering a conditional branch initializes the new frame's d with the test condition's dependencies. During unwinding (Algorithm 1, Line 5), these dependencies propagate upward to parent frames, ensuring all subsequent code paths affected by an abruption inherently depend on its triggering conditions. Furthermore, upon encountering unknown side effects (e.g., writing to an unresolved global), $\Gamma.\text{SideEffect}(d)$ includes both d and the active stack's dependencies in $\mathcal{L}$.

State Updates. Because side effects depend on prior branches, $\Gamma.\text{WriteAddr}$ (Algorithm 2) aggregates both dependencies and control states along the address's allocation path. It attaches the dependencies to the written `DepValue`, making a strong update only if all traversed states are Normal. In Fig. 7, executing `i=2` at ④ invokes $\Gamma.\text{WriteAddr}$. Since ψ_3 is Unsure, the compounded state dictates a weak update (Line 6), yielding `1|2` (⑤). Conversely, unconditional executions like `i=3` trigger strong updates (⑥).

4.1.3 *Non-sequential Control Flow.*

We use *non-sequential control flow* to interpret program constructs with indeterminate execution frequency or timing. Due to the general undecidability of exact modeling, we apply a sound approximation to balance precision and efficiency.

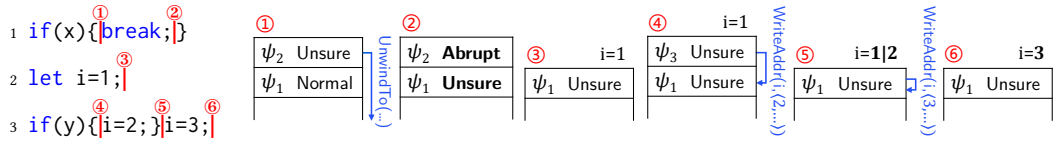

Fig. 7. An example illustrating the control stack and state transitions.

Algorithm 1: Control flow unwinding

```
1 procedure Γ.UnwindTo(κ, d)
2     ψ ← Γ.ψ;   σ ← Abrupt;
3     loop
4         (ψ.σ, σ) ← (σ, ψ.σ ∘ σ); // Propagate state
5         d ← ψ.d ← ψ.d ∪ d; // Propagate deps
6         if ψ.κ = κ then break;
7         ψ ← the parent frame of ψ;
```

Algorithm 2: Write to address

```
1 procedure Γ.WriteAddr(a, ⟨v, d⟩)
2     σ ← Normal;
3     foreach ψ ∈ Path from Γ.ψ to a.ψ do
4         σ ← ψ.σ ∘ σ;   d ← ψ.d ∪ d;
5     if σ = Unsure then
6         a.e ← ⟨v, d⟩ ⊔ a.e; // Weak update
7     else a.e ← ⟨v, d⟩; // Strong update
```

Structural Classification. We classify non-sequential control structures along two orthogonal dimensions: *Iterative* (loops) and *Asynchronous* (non-deterministic invocation timing).

Table 2. Structural classification of non-sequential control flow.

Construct	Example	Iterative	Asynchronous
Unbounded Loops	`while (unknown) { ... }`	✓	–
Unbounded Recursions	`function f() { f(); ... }`	✓	–
Async/Generator Functions	`async function() { ... }`	–	✓
Escaped Functions	`setTimeout(() => { ... })`	✓	✓

Re-interpretation Triggering. During interpretation, we maintain the address sets read (R) and written (W) by each structure. Re-interpretation is triggered by specific memory intersections. For *Iterative* structures, it triggers if the body writes to an address it also reads ($W_{\text{body}} \cap R_{\text{body}} \neq \emptyset$). For *Asynchronous* structures, it triggers whenever an external write targets an address read by the callback ($a_{\text{ext-write}} \in R_{\text{body}}$).

Termination Guarantee. For *Iterative* structures, not every intersecting write triggers re-interpretation. Instead, we compare the incoming abstract value $\langle v', d' \rangle$ against the existing $\langle v, d \rangle$. If subsumed ($v' \sqsubseteq v$), we simply update the address to $\langle v, d \cup d' \rangle$ to capture new dependencies without re-analysis. Otherwise, we update the address with the widened value $\langle v \nabla v', d \cup d' \rangle$ and trigger re-interpretation. The standard widening operator ∇ [Cousot and Cousot 1977] guarantees convergence in finite steps. For *Asynchronous* structures, which are strictly functions, termination is guaranteed by recursion detection (Section 4.3.2).

Efficiency Optimizations. To further minimize re-interpretation, we introduce two mechanisms: (1) *Fast-path for volatile variables.* Variables that previously triggered re-interpretation of an *Iterative* structure are flagged as *volatile*. Subsequent assignments to these variables bypass expensive checks by directly applying the widened value $\langle v_{\text{last}} \nabla v', d' \rangle$ rather than computing the precise $\langle v', d' \rangle$. (2) *Deferred re-interpretation.* We defer the re-interpretation of *Asynchronous* structures to synchronization points (e.g., `await` expressions and unknown function calls). This prevents redundant evaluations during consecutive state updates and aligns with the run-to-completion semantics of the JavaScript event loop.

4.2 Data Flow Analysis

While control flow analysis tracks execution contexts, data flow analysis computes abstract values and propagates dependencies at the expression level.

4.2.1 Evaluation Rules. The evaluation judgment $[ops] \triangleright [\![t^{\ell}]\!] \Downarrow e$ denotes evaluating term t at location ℓ by executing a sequence of operational steps (*ops*) that may produce side effects, ultimately yielding a *DepValue* e. Table 3 summarizes the evaluation of core constructs: Const evaluates to a literal value, and BinOp merges the dependencies from its operands. Variable operations (Read and Write) are resolved against the lexical environment; notably, writing to an unknown global variable triggers an unknown side effect and marks the assigned value as Escaped (Section 4.2.3).

4.2.2 Object Semantics. The core challenge lies in modeling object semantics. As shown in Table 3, ObjLit initializes a record with evaluated key-value pairs. Specifically, it allocates a base address for data properties, while the elided rule for each accessor property Accessor(...) creates or updates an accessor descriptor in the record r by assigning e_i to its π_i component. PropGet

and PropSet handle property reads and writes by first evaluating their sub-terms and looking up relevant descriptors. The helper function ReadDescriptors($v_{\text{obj}}, v_{\text{key}}$) traverses the prototype chain, appending an undefined-valued data descriptor if the property may be absent. Conversely, WriteDescriptors($v_{\text{obj}}, v_{\text{key}}$) resolves L-value descriptors: it locates inherited setters but restricts data descriptors to the receiver object, potentially creating a new data property if one does not exist. For each data descriptor Data(a) in Θ, we directly read the address a or write it via Γ.WriteAddr (Algorithm 2). For each accessor descriptor Accessor($e_{\text{get}}, e_{\text{set}}$), we invoke the corresponding getter or setter via Γ.Call($e_f, \overrightarrow{e_{\text{arg}}}, d$) (Algorithm 3) to capture side effects and the resulting value. Specifically, PropGet invokes the getter without arguments via Γ.Call($e_{\text{get}}, [], \varnothing$), while PropSet passes the assigned value to the setter via Γ.Call($e_{\text{set}}, [e_3], \varnothing$). To track control dependencies and potential nondeterminism, we wrap these operations within control frames. Specifically, if multiple descriptors are possible, the operation is conservatively treated as non-deterministic (Unsure). Dependencies for both the control frame and the resulting *DepValue* are aggregated from the evaluated sub-terms and the expression's location ℓ.

4.2.3 Escaped *Values.* When precise tracking of a value v becomes infeasible (e.g., writes to unknown globals or calls to functions with unknown side effects), the helper function Escaped(v) conservatively over-approximates its potential side effects by traversing v according to its type. To ensure termination, applying Escaped to an already escaped value has no effect. For a union, each constituent value is recursively Escaped. For a record, all descriptor values are Escaped and their dependencies are added to $\mathcal{L}$ to model external accesses, while an unknown property $\top \mapsto \langle \text{Maybe}, \text{Accessor}(\top, \top)\rangle$ is added to model external mutations. For a function, Escaped simulates external invocations via non-sequential control flow (Section 4.1.3) with $\top$ arguments, and its return values are also Escaped. Notably, all values exported by the entry module are Escaped to account for external usage.

Table 3. Evaluation rules for core data flow constructs.

Term t^ℓ	[**Operational Steps** $op_1;\ op_2;\ \dots$] ▷ **Evaluation Result** $[\![t^\ell]\!] \Downarrow e$
Const: v	- ▷ $[\![t^\ell]\!] \Downarrow \langle v, \{\ell\}\rangle$
BinOp: $t_1 \odot t_2$	$[\![t_1]\!] \Downarrow \langle v_1, d_1\rangle;\ [\![t_2]\!] \Downarrow \langle v_2, d_2\rangle;$ ▷ $[\![t^\ell]\!] \Downarrow \langle v_1 \odot v_2, d_1 \cup d_2 \cup \{\ell\}\rangle$
Read: x	$\langle v, d\rangle = x \in \text{dom}(\Gamma.\rho)$ **?** $\Gamma.\rho(x).e$ **:** $\langle \top, \varnothing\rangle;$ ▷ $[\![t^\ell]\!] \Downarrow \langle v, d \cup \{\ell\}\rangle$
Write: $x = t_1$	$[\![t_1]\!] \Downarrow \langle v_1, d_1\rangle$; **if** $(x \in \text{dom}(\Gamma.\rho))$ { Γ.WriteAddr($\Gamma.\rho(x), \langle v_1, d_1 \cup \{\ell\}\rangle$); } **else** { Γ.SideEffect($d_1 \cup \{\ell\}$); Escaped(v_1); } ▷ $[\![t^\ell]\!] \Downarrow \langle v_1, d_1\rangle$
ObjLit: $\{\pi_i[t_{ki}]: t_{vi}{}^{\ell_i}\}$ ($\pi ::= \text{get} \mid \text{set} \mid \epsilon$)	$r = \text{Record}(\texttt{Object.prototype}, \varnothing)$; **foreach** i **do** { $[\![t_{ki}]\!] \Downarrow \langle v_{ki}, d_{ki}\rangle;\ [\![t_{vi}]\!] \Downarrow \langle v_{vi}, d_{vi}\rangle;\ e_i = \langle v_{vi}, d_{ki} \cup d_{vi} \cup \{\ell\}\rangle;$ $r[v_{ki}] = \{\pi_i = \epsilon$ **?** Data($\langle e_i, \Gamma.\psi\rangle$) **:** Accessor($\dots$)$\};$ } ▷ $[\![t^\ell]\!] \Downarrow \langle r, \{\ell\}\rangle$
PropGet*: $t_1[t_2]$	$[\![t_1]\!] \Downarrow \langle v_1, d_1\rangle;\ [\![t_2]\!] \Downarrow \langle v_2, d_2\rangle;\ d = d_1 \cup d_2 \cup \{\ell\};\ \Theta = \text{ReadDescriptors}(v_1, v_2);$ $\langle v_{\text{d}}, d_{\text{d}}\rangle = \bigsqcup\{a.e \mid \text{Data}(a) \in \Theta\};\ e_{\text{d}} = \langle v_{\text{d}}, d_{\text{d}} \cup d\rangle;$ $\Gamma.\text{Push}(\text{PropAccess}, |\Theta| > 1$ **?** Unsure **:** Normal$, d);$ $e_{\text{a}} = \bigsqcup\{\Gamma.\text{Call}(e_{\text{get}}, [], \varnothing) \mid \text{Accessor}(e_{\text{get}}, e_{\text{set}}) \in \Theta\};\ \Gamma.\text{Pop}();$ ▷ $[\![t^\ell]\!] \Downarrow e_{\text{d}} \sqcup e_{\text{a}}$
PropSet*: $t_1[t_2] = t_3$	$[\![t_1]\!] \Downarrow \langle v_1, d_1\rangle;\ [\![t_2]\!] \Downarrow \langle v_2, d_2\rangle;\ [\![t_3]\!] \Downarrow e_3;\ \Theta = \text{WriteDescriptors}(v_1, v_2);$ $\Gamma.\text{Push}(\text{PropAccess}, |\Theta| > 1$ **?** Unsure **:** Normal$, d_1 \cup d_2 \cup \{\ell\});$ **foreach** Data(a) $\in \Theta$ **do** { Γ.WriteAddr(a, e_3); } **foreach** Accessor($e_{\text{get}}, e_{\text{set}}$) $\in \Theta$ **do** { Γ.Call($e_{\text{set}}, [e_3], \varnothing$); } Γ.Pop(); ▷ $[\![t^\ell]\!] \Downarrow e_3$

∗ For briefness, some details like null checking are not formalized.

Algorithm 3: Function invocation

procedure Γ.Call($\langle v_{\text{func}}, d_{\text{func}}\rangle, \overrightarrow{\langle v_{\text{arg}}, d_{\text{arg}}\rangle}, d_{\text{call}}$)
  $e_r \leftarrow \bot$;
  foreach user-defined function $f \sqsubseteq v_{\text{func}}$ **do**
    if occurrences of FuncCall(f) in control stack $\geq$ MaxRecDepth **then**
      Invoke f as unbounded recursions (Section 4.1.3); **continue**;
    Back up current lexical environment $\Gamma.\rho$ and load captured environment $f.\rho$;
    Push control frame: Γ.Push(FuncCall(f), $f \neq v_{\text{func}}$? Unsure : Normal, $d_{\text{func}} \cup d_{\text{call}}$);
    Bind arguments $\overrightarrow{\langle v_{\text{arg}}, d_{\text{arg}}\rangle}$ to parameters $\overrightarrow{f.t_{\text{arg}}}$ and evaluate function body $f.t_{\text{body}}$;
    Pop control frame: Γ.Pop(); Restore $\Gamma.\rho$; Join return *DepValue* to e_r;
  foreach built-in function $\sqsubseteq v_{\text{func}}$ **do**
    Invoke its abstract implementation of f and join result to e_r;
  if call targets contain functions with unknown behavior ($v_{\text{func}} \not\sqsubseteq \top_{\text{F}}$) **then**
    Trigger potential side effects: Γ.SideEffect($d_{\text{func}} \cup \overrightarrow{d_{\text{arg}}} \cup d_{\text{call}}$);
    Mark all arguments as escaped: Escaped($\overrightarrow{v_{\text{arg}}}$); $e_r \leftarrow \top$;
  return e_r;

4.3 Interprocedural Analysis

The first-class nature of JavaScript functions necessitates context-sensitive interprocedural analysis. In this section, we detail function instantiation (Section 4.3.1) and invocation (Section 4.3.2), followed by a function summary mechanism to improve analysis efficiency (Section 4.3.3).

4.3.1 *Function Instantiation.*

Definition 4.6 (User-defined Function Instance f). In the abstract domain, a user-defined function instance is defined as $f ::= \text{Function}(\ell, \overrightarrow{t_{\text{arg}}}, t_{\text{body}}, \rho)$, where ℓ is the definition site for recursion checking, $\overrightarrow{t_{\text{arg}}}$ are the formal parameters, t_{body} is the function body, and ρ is the captured lexical environment enabling context-sensitive analysis.

Evaluating a function definition creates a Function capturing the current lexical environment:

$$\left[\begin{array}{l} v = \text{Function}(\ell, \overrightarrow{t_{\text{arg}}}, t_{\text{body}}, \Gamma.\rho); \\ \textbf{if } (\text{RecInst}(\ell)) \ \{\ \text{Escaped}(v);\ \} \\ \Gamma.\rho(id) \leftarrow \langle\langle v, \{\ell\}\rangle, \Gamma.\psi\rangle; \end{array}\right] \triangleright [\![(\text{function } id(\overrightarrow{t_{\text{arg}}})\ \{t_{\text{body}}\})^{\ell}]\!] \Downarrow \langle v, \{\ell\}\rangle$$

where the predicate $\text{RecInst}(\ell) \iff \ell \in \{f'.\ell \mid \psi \text{ on the control stack}, \ \psi.\kappa = \text{FuncCall}(f')\}$ determines whether the function is being instantiated within its own dynamic extent, and if so, the function is marked as Escaped to prevent precise tracking of infinite function instances.

4.3.2 *Function Invocation.*

The evaluation judgment for function invocation is:

$$\left[\ [\![t_{\text{func}}]\!] \Downarrow e_{\text{func}};\ [\![\overrightarrow{t_{\text{arg}}}]\!] \Downarrow \overrightarrow{e_{\text{arg}}};\ e = \Gamma.\text{Call}(e_{\text{func}}, \overrightarrow{e_{\text{arg}}}, \{\ell\})\ \right] \triangleright [\![(t_{\text{func}}(\overrightarrow{t_{\text{arg}}}))^{\ell}]\!] \Downarrow e$$

Detailed in Algorithm 3, the helper function Γ.Call involves the following steps:

Dynamic Dispatch. To ensure soundness under dynamic dispatch, the analysis conservatively considers all potential targets in v_{func}. For each user-defined target $f \sqsubseteq v_{\text{func}}$, it performs a context-sensitive call, whereas built-in targets invoke their abstract implementations directly. If targets

include functions with unknown behavior ($v_{\text{func}} \not\sqsubseteq \top_{\text{F}}$), we conservatively trigger unknown side effects via Γ.SideEffect, mark all arguments as Escaped and set the return value to $\top$.

Recursion Detection. To ensure termination, we check the occurrences of the invoked function instance in the active control stack. If the count reaches MaxRecDepth, we treat the invocation as an unbounded recursion and apply the non-sequential control flow strategy (Section 4.1.3). As evaluated in Section 6, setting MaxRecDepth to 2 strikes an optimal balance between accommodating common patterns and controlling computational overhead.

Context-Sensitive Execution. For user-defined targets, we switch the lexical environment $\Gamma.\rho$ to the captured scope $f.\rho$ and push a new control frame of kind FuncCall(f). If the call target is ambiguous ($f \neq v_{\text{func}}$), the frame is treated as non-deterministic (Unsure). After binding actual arguments to formal parameters and evaluating the function body, we pop the frame and restore the original lexical environment, joining the result into the aggregated return value e_r.

Return Statement Handling. Upon a `return` statement, we collect the returned value by writing it to a special variable `"ret"` (pre-allocated in the function's lexical scope and control frame with an initial value of undefined). This leverages the existing Γ.WriteAddr logic to capture control dependencies for conditional returns automatically. Then, an Γ.UnwindTo operation targeting the closest function call frame propagates the control state and dependencies. The statement itself yields no value and thus evaluates to Ok:

$$\left[\begin{array}{l}[\![t]\!] \Downarrow \langle v, d\rangle;\ \Gamma.\text{WriteAddr}(\Gamma.\rho(\texttt{"ret"}), \langle v, d \cup \{\ell\}\rangle);\\ \Gamma.\text{UnwindTo}(\text{FuncCall}(*), \{\ell\}); // * \textit{ matches any function instance}\end{array}\right] \triangleright [\![(\text{return } t)^{\ell}]\!] \Downarrow \text{Ok}$$

4.3.3 Function Summary. Function summary in JsShaker is an opt-in mechanism that significantly enhances analysis efficiency by reusing analysis results of called functions.

Symbolic Dependency. Unlike general function summary, our approach must be *dependency-aware*. However, dependencies are too costly to serialize, and their inclusion states remain latent until the completion of the entire analysis. To address this, we introduce *Symbolic Dependency* τ as a new dependency variant, alongside a *Latent Dependencies Set* $\mathcal{Z}$ that links each τ to its concrete dependencies. As formalized below, if a symbolic dependency is included in $\mathcal{L}$, all its linked dependencies are also included.

$$\begin{aligned}(\textit{Dependency}) \quad & \delta ::= \cdots \mid \tau\\ (\textit{SymbolDependency}) \quad & \tau ::= i \text{ (unique identifier)}\\ (\textit{Latent Dependencies Set}) \quad & \mathcal{Z} ::= \{\langle \tau \mapsto d\rangle\}\end{aligned} \qquad \frac{\langle \tau \mapsto d\rangle \in \mathcal{Z} \quad \tau \in \mathcal{L}}{d \subseteq \mathcal{L}}$$

Summary Structure. For each context-aware function instance f, the analysis maintains a set of summary entries: $\eta ::= \{\langle \overrightarrow{v_{\text{arg}}}, \overrightarrow{\tau_{\text{arg}}}, e_{\text{ret}}, r, w, g\rangle\}$, where $\overrightarrow{v_{\text{arg}}}$ and $\overrightarrow{\tau_{\text{arg}}}$ denote the abstract arguments and their corresponding symbolic dependencies, and e_{ret} is the returned *DepValue*. To capture non-local interactions, the summary records external reads and writes: $r : \textit{Address} \to (\textit{AbstractValue} \times \textit{SymbolDependency})$ maps external read addresses to their values and symbolic dependencies, while $w ::= \overrightarrow{\langle a, e, \sigma\rangle}$ is a sequence of external writes as (*Address*, *DepValue*, *ControlState*) triples. Finally, the boolean flag g indicates the presence of unknown side effects. In implementation, hash maps index these summaries by their arguments: $\overrightarrow{v_{\text{arg}}} \to \langle \overrightarrow{\tau_{\text{arg}}}, e_{\text{ret}}, r, w, g\rangle$.

Summary Matching. Upon calling a function f, JsShaker seeks a summary entry on f where the arguments v_{arg} match, and the recorded non-local reads r are compatible with the current context.

Summary Generation. If no matching summary exists, the function body is evaluated via Algorithm 3 to generate a new entry: (1) *Argument Initialization*: Pass the arguments, each augmented with a new symbolic dependency τ_{arg}, to Algorithm 3. (2) *Function Evaluation*: When evaluating the body, all non-local interactions are recorded into r and w, while the read values are attached with new symbolic dependencies. If the return value or non-local writes involve objects allocated within the function call, the generation is aborted. (3) *Finalization*: Store the summary entry with the return *DepValue* e_{ret} and the side-effect flag g (set if Γ.SideEffect was invoked).

Summary Application. If a matching summary is found, apply it as follows: (1) *Link Current Dependencies as Latent Dependencies*: $\mathcal{Z} \leftarrow \mathcal{Z} \cup \{\langle \tau_{\text{arg}i} \mapsto e_{\text{arg}i}.d \rangle\} \cup \{\langle \tau \mapsto a.e.d \rangle \mid r(a) = \langle v, \tau \rangle\}$. (2) *Applying External Writes*: For each $\langle a, e, \sigma \rangle \in w$, perform Γ.WriteAddr(a, e) under the control state σ. (3) *Side Effect Triggering*: Invoke Γ.SideEffect$(\varnothing)$ if the flag g is set. (4) *Result Projection*: Return the recorded *DepValue* e_{ret}.

Trade-Off Between Precision and Efficiency. Our function summaries inherently trade analysis precision for efficiency. The design above balances this in two ways: (1) to mitigate the loss of context-sensitivity, argument values are matched invariantly during summary retrieval; and (2) to preserve object-sensitivity, the return value and non-local writes must not involve objects allocated within the function call.

5 Framework Extensibility: Advanced Optimizations

Building on the single-round analysis and transformation framework above, this section demonstrates three advanced optimizations enabled by new dependency variants and liveness analysis: branch folding (Section 5.1), constant folding (Section 5.2), and property mangling (Section 5.3).

5.1 Branch Folding

While foundational tree-shaking removes unreachable branches, *Branch Folding* eliminates deterministic conditional checks and their predicate code.

Call Site Tracking. A non-deterministic condition may become deterministic after pruning some call sites to its enclosing function. For instance, in Listing 1, while `if(x)` is initially non-deterministic, it becomes constant falsy after eliminating the first invocation `f("42")` via constant folding. To handle such cases, the *CallSite* $c ::= \ell$ is tracked as $\Gamma.c$, updated upon each invocation. For top-level code, a special call site $c_{\text{entry}} \in \mathcal{L}$ represents the entry point of the program.

Evaluation of Conditional Constructs. Every execution of a conditional construct is recorded in the global set of *Entered Branches* $\mathcal{B} ::= \{\langle \ell, c, v, d \rangle\}$, where ℓ uniquely identifies the construct, c is the triggering call site, and v and d are the abstract value and dependency set of the evaluated test condition, respectively. Upon encountering such a construct, $\mathcal{B}$ is extended, and ℓ serves as a control dependency to signify the purity of the entire construct:

$$\left[\begin{array}{l} [\![t_1]\!] \Downarrow \langle v, d \rangle;\ \mathcal{B} = \mathcal{B} \cup \{\langle \ell, \Gamma.c, v, d \rangle\}; \\ \sigma = (v = \top_{\text{Bool}})\ \texttt{?}\ \text{Unsure}\ \texttt{:}\ \text{Normal}; \\ \textbf{if}\ (\text{True} \sqsubseteq v)\ \{\ \Gamma.\text{Push}(\text{Branch}, \sigma, \{\ell\});\ [\![t_2]\!] \Downarrow \text{Ok};\ \Gamma.\text{Pop}();\ \} \\ \textbf{if}\ (\text{False} \sqsubseteq v)\ \{\ \Gamma.\text{Push}(\text{Branch}, \sigma, \{\ell\});\ [\![t_3]\!] \Downarrow \text{Ok};\ \Gamma.\text{Pop}();\ \} \end{array}\right] \triangleright\ [\![\text{if}^{\ell}(t_1)\ t_2\ \text{else}\ t_3]\!] \Downarrow \text{Ok}$$

Deciding Determinism. To determine whether a conditional check is redundant, the execution records in $\mathcal{B}$ are filtered against the *Liveness Set* $\mathcal{L}$. This derives the *Non-Deterministic Conditionals* $\mathcal{N} ::= \{\ell\}$, which consist of impure conditional constructs ($\ell \in \mathcal{L}$) where both paths are traversed across preserved execution contexts ($\{c_1, c_2\} \subseteq \mathcal{L}$). In such cases, the test condition dictates the

control flow and is deemed live ($d_1 \cup d_2 \subseteq \mathcal{L}$):

$$\frac{\langle \ell, c_1, v_1, d_1\rangle \in \mathcal{B} \quad \langle \ell, c_2, v_2, d_2\rangle \in \mathcal{B} \quad \{\ell, c_1, c_2\} \subseteq \mathcal{L} \quad \mathsf{True} \sqsubseteq v_1 \quad \mathsf{False} \sqsubseteq v_2}{\ell \in \mathcal{N} \quad d_1 \cup d_2 \subseteq \mathcal{L}}$$

Transformation. The conditional check is eliminated if it is deterministic ($\ell \notin \mathcal{N}$):

$$\mathrm{if}^{\ell}\,\widehat{(t_1)\ t_2\ \mathrm{else}}\ t_3 = \begin{cases} \mathrm{if}\ (\widehat{t_1})\ \widehat{t_2}\ \mathrm{else}\ \widehat{t_3}, & \text{if } \ell \in \mathcal{N} \\ \widehat{t_1};\ \widehat{t_2};\ \widehat{t_3}, & \text{otherwise} \end{cases}$$

where $\widehat{t_2}$ or $\widehat{t_3}$ is reduced to empty if that branch is unreachable or pure, as defined in Section 3.4.

5.2 Constant Folding

Constant folding reduces code size by collapsing runtime computations into literals. By integrating dependency-aware abstract values, JsShaker ensures that eliminated code paths do not interfere with constant folding results. This reveals further dead code, all in a single analysis round.

Folding Requests. A folding request $\mathsf{Fold}(\ell, v, d)$ is added as a new dependency variant of δ in Fig. 5, whose liveness indicates that either term at ℓ is replaced by v in output, or original dependencies d (used to compute v) must be included. As defined in Eq. (1) of Fig. 8, the original evaluation $[\![t]\!]^{\mathrm{base}}$ is wrapped for all expressions to attach folding requests.

Constant Resolution. Formalized in Eq. (2) of Fig. 8, *Constant Candidate Map* $\mathcal{K}$: *CodeLocation* $\rightarrow$ *AbstractValue* is computed as the join of values from all included folding requests, effectively preventing dead code paths from polluting the result. If $\mathcal{K}(\ell)$ resolves to a literal value, the term at ℓ can be safely folded. Otherwise, if the result is not a literal (due to conflicting values or non-constant results), Eq. (3) dictates that the original computation must be preserved by including the original dependencies d. Because the inclusion of d is bypassed when the term at ℓ is folded, this mechanism naturally prunes the underlying computation paths, thereby reinforcing tree-shaking.

Transformation. As defined in Eq. (4) of Fig. 8, $\mathcal{K}$ dictates the transformation strategy for each expression. Here, the comma expression $(\widehat{t}^{\mathrm{base}}, \mathcal{K}(\ell))$ preserves any necessary side effects by retaining $\widehat{t}^{\mathrm{base}}$, the transformation without constant folding.

5.3 Property Mangling

Property mangling offers code size reduction opportunities; current tools typically rely on unsound heuristics or manual intervention for dynamic accesses. Prioritizing correctness over fragile optimizations, JsShaker casts conservative, automatic mangling as a constraint-solving problem.

Property Mangling as String Mangling. To unify the analysis model, all static property names—whether in object literals (e.g., `{ prop: val }`) or member expressions (e.g., `obj.prop`)—are treated as computed property accesses using string literals (e.g., `obj["prop"]`). Consequently, the problem of property mangling is reduced to tracking the flow and constraints of string values.

$$\left[\ [\![t]\!]^{\mathrm{base}} \Downarrow \langle v, d\rangle\ \right] \triangleright [\![t^{\ell}]\!] \Downarrow \langle v, \{\mathsf{Fold}(\ell, v, d)\}\rangle \quad (1)$$

$$\mathcal{K}(\ell) \triangleq \bigsqcup \{v \mid \exists d.\ \mathsf{Fold}(\ell, v, d) \in \mathcal{L}\} \quad (2)$$

$$\frac{\mathcal{K}(\ell) \text{ is not a literal} \quad \mathsf{Fold}(\ell, v, d) \in \mathcal{L}}{d \subseteq \mathcal{L}} \quad (3)$$

$$\widehat{t^{\ell}} = \begin{cases} (\widehat{t}^{\mathrm{base}}, \mathcal{K}(\ell)), & \text{if } \mathcal{K}(\ell) \text{ is a literal} \\ \widehat{t}^{\mathrm{base}}, & \text{otherwise} \end{cases} \quad (4)$$

Fig. 8. Core rules for constant folding: requests (1), resolution (2), liveness (3), and transformation (4).

Mangling Candidate String. The abstract domain $\mathcal{V}$ is extended with $\mathsf{MglStr}(s, \ell)$, where s is the string content and ℓ identifies its creation site to map generated constraints back to AST nodes:

$$\mathcal{V} \ni \mathsf{MglStr}(s, \ell) \qquad [\ ] \rhd [\![s^{\ell}]\!] \Downarrow \langle \mathsf{MglStr}(s, \ell), \{\ell\} \rangle$$

Mangling Constraints. We extend δ in Fig. 5 with three dependency variants for safe property mangling: $\delta ::= \cdots \mid \ell_1 \sim \ell_2 \mid \ell_1 \nsim \ell_2 \mid \ell \equiv s$, where

- **Must-Alias** ($\ell_1 \sim \ell_2$): Strings originating at ℓ_1 and ℓ_2 must be mangled to the same string.
- **Must-Diff** ($\ell_1 \nsim \ell_2$): Strings originating at ℓ_1 and ℓ_2 must be mangled to distinct strings.
- **Pinned** ($\ell \equiv s$): The string originating at ℓ must be preserved exactly as s.

Constraint Generation. Operations involving strings utilize two functions to generate constraints:

- $\mathsf{Pin}(v) \triangleq \{\ell \equiv s \mid \mathsf{MglStr}(s, \ell) \sqsubseteq v\}$. This function generates constraints preventing the mangling of any string in v.
- $\mathsf{Compare}(v_1, v_2) \triangleq \begin{cases} \left\{ (s_1 = s_2)\ ?\ \ell_1 \sim \ell_2 : \ell_1 \nsim \ell_2 \;\middle|\; \begin{smallmatrix} \mathsf{MglStr}(s_1, \ell_1) \sqsubseteq v_1 \\ \mathsf{MglStr}(s_2, \ell_2) \sqsubseteq v_2 \end{smallmatrix} \right\}, & \text{if } v_1, v_2 \sqsubseteq \top_{\mathsf{M}} \\ \mathsf{Pin}(v_1) \cup \mathsf{Pin}(v_2), & \text{otherwise} \end{cases}$.

  This function generates minimal constraints required to preserve the comparison results. If both values resolve to mangling candidate strings ($v_1, v_2 \sqsubseteq \top_{\mathsf{M}}$, where $\top_{\mathsf{M}}$ is the join of all MglStr), they can be safely mangled under these conditional constraints. Otherwise, they are conservatively pinned.

Operations that compare strings, including equality comparison and Read/WriteDescriptors, utilize Compare to generate aliasing constraints, enabling potential safe mangling:

$$\left[[\![t_1]\!] \Downarrow \langle v_1, d_1 \rangle;\ [\![t_2]\!] \Downarrow \langle v_2, d_2 \rangle \right] \rhd [\![(t_1 \mathrel{===} t_2)^{\ell}]\!] \Downarrow \langle v_1 \stackrel{?}{=} v_2, d_1 \cup d_2 \cup \mathsf{Compare}(v_1, v_2) \cup \{\ell\} \rangle$$

Conversely, for non-comparison operations—such as string concatenation or type coercions—$\mathsf{Pin}(v)$ is conservatively added to the result's dependencies to prevent unsafe mangling. Crucially, $\mathsf{Escaped}(v)$ must enforce $\mathsf{Pin}(v) \subseteq \mathcal{L}$.

Constraint Solving. With $\mathcal{L}$ established from the analysis phase, a simple greedy solver computes the mangling map, $\mathcal{M} : \mathit{CodeLocation} \to \mathit{String}$, assigning the shortest valid identifier to each creation site ℓ while satisfying all constraints included in $\mathcal{L}$.

Transformation. Working independently, transforming a string literal s^{ℓ} yields its mangled version: $\widehat{s^{\ell}} = \mathcal{M}(\ell)$. When composed with constant folding, this extends to any expression folding to a MglStr: $\widehat{t^{\ell}} = \begin{cases} (\widehat{t}^{\,\mathrm{base}}, \mathcal{M}(\ell')), & \text{if } \mathcal{K}(\ell) = \mathsf{MglStr}(s, \ell') \\ \widehat{t}^{\,\mathrm{CF}}, & \text{otherwise} \end{cases}$, where $\widehat{t}^{\,\mathrm{CF}}$ denotes the transformation defined by constant folding.

6 Evaluation

We conduct evaluations to answer the following research questions:

RQ1 What code size reduction does JsShaker achieve across module imports and applications, and how do specific mechanisms (e.g., property mangling) contribute to this reduction?

RQ2 What build-time overhead does JsShaker introduce, how does it scale across massive codebases, and what is the impact of function summaries on this performance?

RQ3 Does JsShaker safely preserve JavaScript execution semantics?

Implementation. We implement JsShaker in Rust based on the Oxc parser and code generator [Chen et al. 2026]. For seamless integration, we provide a CLI, Node.js/WebAssembly NAPI-RS bindings [Long et al. 2026], and a Rollup plugin [Harris et al. 2026].

Table 4. Overview of the dataset. The target programs are constructed by importing representative functions from 13 widely used JavaScript projects across various domains.

Category	Project (Target)	kLOC	Size (KB)	#File	#Pkg	Description	Stars	Date
Data Parsing	js-yaml (load)	2.8	105	2	1	YAML parser and serializer	6,557	2025-11
File System	glob (globSync)	5.0	245	19	7	File path pattern matching utility	8,722	2026-02
Functional	remeda (pipe,range,filter)	1.3	39	62	1	TypeScript-first functional utils	5,317	2026-02
Icons	react-icons (FaBeer)	6.9	1,356	8	2	SVG icon components for React	12,519	2026-03
Monitoring	sentry (init)	32.0	1,520	261	6	Error tracking and monitoring SDK	8,602	2026-03
Networking	novnc (UI.start)	13.6	650	58	1	VNC client built with HTML5	13,541	2026-01
Presentation	slidev-demo (frontend)	227.1	8,291	412	54	Slides framework for developers	44,850	2026-03
Reactivity	rxjs (range,map,filter)	8.5	332	201	2	Observable-based reactivity library	31,655	2025-02
UI Library	antd (DatePicker)	120.3	4,427	1,422	67	Enterprise-grade React.js UI	97,696	2026-03
UI Library	material-ui (Paper)	39.4	1,758	487	26	Material Design for React.js	98,003	2022-04
UI Library	vuetify (VApp,VBtn)	49.3	1,677	434	6	Component framework for Vue.js	40,998	2026-03
Utilities	lodash-es (flowRight,range,filter)	9.3	613	639	1	ESM build of the Lodash utilities	61,578	2026-01
Visualization	d3 (select,scaleLinear)	19.9	680	555	34	Data-driven DOM manipulation	112,538	2024-03
Total (deduped)		**533.9**	**21,645**	**4,530**	**195**			

Dataset. We select 13 JavaScript libraries across diverse categories, as detailed in Table 4. To simulate real-world tree-shaking scenarios, we construct target programs for each library by importing representative functions or components.

Experimental Setup. Evaluations are conducted on an Ubuntu 24.04 workstation (Intel Core Ultra 9 285K, 24-core, 32GB RAM) running Node.js v24.11.1. To simulate a standard JavaScript pipeline, we use Rollup v4.57.1 [Harris et al. 2026] (tree-shaking enabled) for bundling and Terser v5.46.0 [Maiwald et al. 2026] for minification. We evaluate JsShaker (using a MaxRecDepth = 2 by default) against Google Closure Compiler v20260128 [Google 2026] (Simple/Advanced modes, denoted $CC_{Simp.}$ and $CC_{Adv.}$) and Lacuna [Malavolta et al. 2023] (analyzer `{dynamic:0.5,jelly:0.5}`, levels O2/O3, omitting O1 as it inflates size). We insert these tools between Rollup and Terser to evaluate their impact on code size and build time. DFAHC [Farzat et al. 2021] is excluded due to its prohibitive build time.

6.1 RQ1: Code Size Reduction

6.1.1 Comparison with Baselines. We evaluate code size reduction against the Rollup + Terser (R+T) baseline, measuring sizes post-Gzip (level 6) to simulate Web transmission. Table 5 details the relative reduction percentages. We exclude invalid runs, specifically those that failed at build time or broke semantic equivalence. As shown, JsShaker consistently outperforms all alternatives, achieving the highest valid size reduction across all 13 target programs, with a maximum reduction of 66.7% (react-icons). The (React-aware) column shows the results when JsShaker's opt-in React-awareness[1] is enabled, further pushing the average reduction from 27.6% to 30.2%.

Closure Compiler in Simple mode successfully optimized all programs except glob (due to its lack of Node.js support), but none of its results surpassed JsShaker. Advanced mode yielded valid outputs for only four programs, primarily due to its aggressive but unsafe property mangling, with a best valid reduction of 30.4% (lodash-es). Turning to Lacuna, both its O2 and O3 modes frequently failed: O2's valid runs never outperformed JsShaker, and O3 yielded no valid output at all.

In some cases (material-ui, novnc, vuetify), Closure Compiler's Simple mode increased the final size by rendering Terser less effective. Both JsShaker and Closure Compiler may marginally increase

[1]This mode incorporates abstract implementations of React built-ins. However, as it currently requires modifying the standard build pipeline, it is not enabled in subsequent evaluations.

Table 5. Post-Gzip code size reduction vs. the Rollup + Terser baseline (higher is better). **Bold**: highest reduction among valid runs; *Italic*: failed (e.g., *Syntax Err.*); ~~Strikethroughs~~: incorrect semantics.

Program	Baseline (KB)	JsShaker (React-aware)	$CC_{Simp.}$	$CC_{Adv.}$	$Lacuna_{O2}$	$Lacuna_{O3}$
antd	165.20	**4.5%** (18.7%)	0.1%	~~5.6%~~	*Syntax Err.*	~~90.0%~~
d3	15.55	**52.8%**	0.1%	14.1%	*Syntax Err.*	~~42.5%~~
glob	25.67	**21.7%**	*Unsupported*	*Unsupported*	*Syntax Err.*	*Syntax Err.*
js-yaml	12.53	**27.0%**	24.9%	~~26.5%~~	~~38.6%~~	~~38.1%~~
lodash-es	6.37	**53.8%**	28.6%	30.4%	2.4%	~~2.3%~~
material-ui	69.56	**38.9%** (40.5%)	−0.2%	~~5.4%~~	~~33.3%~~	~~33.3%~~
novnc	60.09	**4.2%**	−0.8%	~~8.8%~~	~~83.4%~~	*Syntax Err.*
react-icons	3.66	**66.7%** (74.9%)	0.1%	~~15.0%~~	33.1%	~~−181.6%~~
remeda	0.92	**9.6%**	2.0%	6.5%	0.4%	~~0.5%~~
rxjs	2.64	**16.1%**	1.9%	10.8%	~~88.9%~~	*Syntax Err.*
sentry	24.92	**4.9%**	2.9%	~~12.4%~~	*Syntax Err.*	*Syntax Err.*
slidev-demo	971.62	**6.5%**	1.4%	~~6.9%~~	*OOM*	*OOM*
vuetify	43.78	**3.7%**	−0.1%	~~8.6%~~	*Syntax Err.*	*Syntax Err.*
Geomean		**27.6%** (30.2%)	**5.6%**	**16.0%**	**13.4%**	**N/A**

$CC_{Simp.}$ and $CC_{Adv.}$ denote Google Closure Compiler v20260128 in Simple/Advanced modes.

the uncompressed raw code size by folding single constants into multiple literals. This intentional redundancy is highly compressible by Gzip, effectively minimizing the final transmitted size.

6.1.2 Ablation Study. Fig. 9 visualizes the relative impact of our advanced optimizations. With tree-shaking serving as JsShaker's foundation, we isolate each advanced technique—constant folding, branch folding, and property mangling—by measuring the additional size reduction it achieves when enabled alongside foundational tree-shaking. These gains are then normalized against the total reduction from the fully optimized configuration. Note that due to the synergistic nature of compiler optimizations, these isolated percentages are approximate.

As shown, foundational tree-shaking consistently drives the bulk of the size reduction, contributing 72.3% on average. Building upon this, constant folding and branch folding deliver highly variable yet substantial additional gains, averaging 12.7% and 12.0% of the total optimization, respectively. Property mangling plays a supplementary role with a 3.0% average contribution, demonstrating the practical viability of safe property mangling.

Notably, the contribution of each strategy is highly sensitive to the target library's architectural patterns. For instance, in react-icons, the optimization almost exclusively targets unused SVG icons, allowing foundation tree-shaking to achieve 100% of the reduction. Conversely, advanced

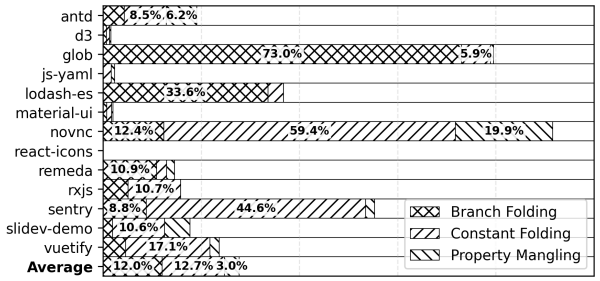

Fig. 9. Ablation study of size reduction by JsShaker's optimization strategies. Foundational tree-shaking (white segments) establishes the foundation, while advanced optimizations (hatched segments) vary across programs.

Table 6. Comparison of property mangling rates between JsShaker and Closure Compiler.

Program	Total	$CC_{Adv.}$ (%)	JsShaker (%)	JsShaker/ $CC_{Adv.}$(%)
antd	19,158	41.38	4.31	10.42
d3	1,040	35.79	7.02	19.61
js-yaml	1,031	19.21	3.43	17.84
lodash-es	251	13.00	3.23	24.82
material-ui	6,255	47.65	3.11	6.52
novnc	8,354	61.08	12.42	20.33
react-icons	201	47.92	15.73	32.82
remeda	80	42.86	4.00	9.33
rxjs	301	40.51	0.00	0.00
sentry	2,549	44.23	3.79	8.57
vuetify	4,590	39.32	3.37	8.58
Average		**39.36**	**5.49**	**13.95**

optimizations dominate in highly self-contained codebases like novnc: abundant constant definitions drive constant folding (59.4%), fixed initialization parameters enable branch folding to prune dead paths, and explicit class structures push property mangling to its peak contribution of 19.9%.

6.1.3 Mechanism Deep Dive: Safe Property Mangling. Table 6 evaluates JsShaker's safe property mangling against the aggressive but unsafe mangling of Closure Compiler in Advanced mode ($CC_{Adv.}$). $CC_{Adv.}$ unsafely mangles 39.36% of properties on average, establishing an approximate upper bound for property renaming. JsShaker achieves a safe mangling rate of 5.49%, which accounts for 13.95% of this upper bound. These results demonstrate the effectiveness of our approach, yet significant room remains to fully realize the potential of safe property mangling.

6.2 RQ2: Build Time Overhead

6.2.1 Comparison with Baselines. Table 7 reports the build time overhead of each optimizer. Absolute baseline execution times (Rollup + Terser, R+T) are provided in milliseconds, while all other results—including Rolldown (RD) [VoidZero Inc. 2026] as a Rust-based reference—are presented as multipliers (×) of this baseline. JsShaker and Lacuna are timed across the entire Rollup + Optimizer + Terser pipeline. Because Closure Compiler integrates minification, it is timed simply as Rollup + CC. Overall averages represent the geometric mean of these multipliers across successful runs.

JsShaker demonstrates a low overhead, increasing the total build time to an average of 1.45× of the baseline. Its peak overhead (1.98× on `js-yaml`) arises from the analysis of complex recursive code structures. Closure Compiler incurs moderate delays, averaging 2.49× and 4.48× in Simple and Advanced modes, respectively. Lacuna introduces substantial overhead, averaging 21.9× in both O2 and O3 modes due to the computational complexity of its underlying static analysis.

6.2.2 Sensitivity of MaxRecDepth. Fig. 10 evaluates the impact of varying MaxRecDepth. Increasing the depth from 1 to 2 yields a measurable improvement in code size reduction (from 26.55% to 27.58%). However, further increasing the depth up to 5 yields negligible additional reduction (< 0.01%) while the average build time continues to rise significantly (from 624.07 ms at depth 2 to

Table 7. Build time overhead. Baseline (R+T:Rollup+Terser) execution times are in milliseconds, while other results are presented as multipliers (×) of the baseline. Rolldown (RD) serves as a Rust-based reference. Best results are bolded; "—" denotes build-time failures.

Project	With Baseline		Optimizer (×)				
	R+T (ms)	RD (×)	JsShaker	$CC_{Simp.}$	$CC_{Adv.}$	$Lacuna_{O2}$	$Lacuna_{O3}$
antd	2952	0.03	**1.27**	1.41	1.76	17.5	17.5
d3	321	0.05	**1.01**	2.07	3.29	13.2	13.1
glob	145	0.09	**1.05**	—	—	30.0	29.6
js-yaml	70	0.11	**1.98**	5.96	12.2	54.8	55.3
lodash-es	316	0.04	**1.00**	2.02	3.10	12.6	12.7
material-ui	921	0.04	**1.05**	1.65	2.27	11.8	11.9
novnc	353	0.07	**1.11**	2.40	4.30	14.1	14.0
react-icons	145	0.15	**1.02**	4.13	7.30	41.0	40.9
remeda	29	0.10	**1.01**	9.66	23.3	124	125
rxjs	150	0.03	**1.00**	2.67	5.46	25.2	25.5
sentry	406	0.04	**1.02**	1.82	3.34	11.3	11.3
slidev-demo	13075	—*	1.54	**1.41**	—	—	—
vuetify	801	0.06	**1.06**	1.52	2.20	10.4	10.5
Geomean	—	**0.06**	**1.14**	**2.49**	**4.48**	**21.9**	**21.9**

∗ slidev-demo is incompatible with Rolldown.

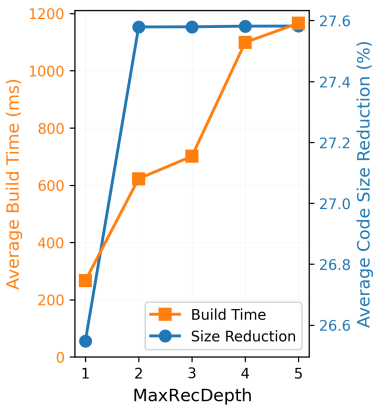

Fig. 10. Impact of MaxRecDepth on build time and code size. Beyond depth 2, build time increases with negligible size reduction.

Table 8. Performance impact of function summaries. By mitigating exponential analysis blowup in complex programs, the summary mechanism yields an overall 1.58× geometric mean speedup. Minor regressions occur only when caching overhead exceeds the cost of directly evaluating trivial functions.

Project	#FunDef	#AbsInst	#Calls	Retrieval (%)			Generation (%)			Time (ms)		Speedup (×)
				ArgMiss	EnvMiss	Hit	PropRW	ObjRet	Updated	w/o Sum.	w/ Sum.	
antd	3,294	15,390	193,757	61.98	0.06	37.96	60.35	27.63	12.01	18725.03	804.41	23.28
d3	372	410	603	57.55	3.15	39.30	39.07	14.75	46.17	4.76	3.53	1.35
glob	525	619	1,129	77.41	1.51	21.08	41.41	3.14	55.44	8.00	7.79	1.03
js-yaml	144	183	26,697	50.56	0.00	49.44	62.75	36.14	1.11	79.54	90.69	0.88
lodash-es	154	155	202	68.81	0.00	31.19	25.90	9.35	64.75	1.61	1.46	1.10
material-ui	789	933	3,317	51.46	0.84	47.69	24.21	18.50	57.29	94.55	55.69	1.70
novnc	699	776	9,092	61.00	0.02	38.98	74.32	2.47	23.22	76.05	44.17	1.72
react-icons	1,625	1,629	35	100.00	0.00	0.00	68.57	25.71	5.71	3.52	3.54	0.99
remeda	23	27	36	100.00	0.00	0.00	44.44	5.56	50.00	0.37	0.37	0.99
rxjs	92	97	147	81.63	0.68	17.69	24.79	10.74	64.46	0.71	0.73	0.98
sentry	639	920	2,365	67.23	0.68	32.09	49.44	7.78	42.78	15.76	9.06	1.74
slidev-demo	17,153	151,659	1,264,575	73.74	0.15	26.11	67.49	14.51	17.99	10046.96	7045.21	1.43
vuetify	1,000	5,785	19,545	74.80	0.46	24.74	45.11	4.47	50.42	81.05	46.19	1.75
Average				**71.71**	**0.14**	**28.15**	**66.24**	**16.01**	**17.74**	**2241.38**	**624.07**	**1.58**

1166.04 ms at depth 5). This confirms that defaulting MaxRecDepth to 2 strikes the optimal balance between practical optimization and computational overhead.

6.2.3 Efficacy of Function Summaries. Table 8 assesses the performance impact of function summaries via overall speedup and analysis of retrieval/generation failures across all function calls.

The table reports the total number of reached user function definitions (#FunDef), their context-bound abstract instances (#AbsInst), and invocations (#Calls). Summaries are reused (Hit) only when abstract arguments and environmental memory match; mismatches cause ArgMiss or EnvMiss. On miss, functions are dynamically evaluated; new summaries are then saved (Updated) unless the evaluation interacts with external properties (PropRW) or returns object references (ObjRet). We currently skip summary generation for the PropRW case to bound implementation complexity, whereas the ObjRet case is deliberately excluded to preserve object-sensitivity.

Our evaluation reveals ArgMiss (avg. 71.71%) and PropRW (avg. 66.24%) as the primary bottlenecks for summary retrieval and generation, respectively. Addressing PropRW presents a clear improvement path: better tracking of property mutations could significantly boost the update rate and overall speed. Ultimately, the mechanism yields a 1.58× geometric mean speedup (peaking at 23.28× for antd), with minor regressions in projects featuring structurally trivial functions (e.g., js-yaml due to low update rate).

6.3 RQ3: Standards Compliance and Correctness

To evaluate JsShaker's correctness and compliance with language standards, in addition to manually verifying the target programs in Table 4, we utilized the official Test262 suite [Ecma TC39 2026]. Out of the 48,747 total test cases, we excluded 18.79% due to engine262 [engine262 Contributors 2026] incompatibilities (primarily unsupported stage proposals) and 14.16% for violating JsShaker's analysis assumptions. The remaining 32,688 cases (67.06%) underwent full optimization under a conservative configuration (e.g., treating unknown global variable reads as side effects), and passed when run in engine262. This 100% pass rate demonstrates JsShaker's ability to safely analyze and optimize modern JavaScript features while strictly preserving original execution semantics.

7 Discussion

Impact on Runtime Efficiency. By preserving the operational semantics and AST structure of live code, JsShaker introduces no runtime overhead. Instead, it yields marginal performance improvements: reduced code size alleviates parsing and JIT compilation, property mangling accelerates dynamic lookups via shorter names, and constant folding eliminates redundant computations.

Correctness of JsShaker. JsShaker ensures semantics preservation through accurate abstract interpretation and comprehensive dependency tracking. For determinable control and data flows, our analysis precisely computes abstract states to capture all behaviorally relevant dependencies. Crucially, for ambiguous JavaScript constructs that defy precise static resolution, we adopt a conservative over-approximation strategy to ensure that any behaviorally relevant logic is retained. This theoretical safety is empirically validated by our 13 target programs and Test262 evaluation, confirming that JsShaker introduces no behavioral regressions across supported language features.

Limitations of JsShaker. Although context-sensitive, JsShaker forgoes path-sensitivity to avoid maintaining separate dependency-aware abstract states along divergent branches. This necessary approximation may occasionally miss optimizations. Moreover, consistent with our *soundiness* goal, we provide best-effort handling rather than strict guarantees for certain analysis-defeating dynamic features: (1) non-strict mode; (2) lexical-scope-dependent dynamic code evaluation (e.g., `eval`, whereas `new Function` is supported); (3) control flows from implicit runtime exceptions (e.g., property accesses on `undefined`); (4) reliance on `func.name` and `func.length` properties; and (5) built-in object prototype mutations. Approximating these edge cases is a standard, pragmatic compromise in static analysis that rarely affects well-formed, modern JavaScript programs.

Extending Framework-Aware Optimizations. Consistent with our architectural adaptability goal, JsShaker features a framework-agnostic core that seamlessly accommodates optimizations tailored to libraries like React and Vue.js. Providing abstract API definitions, dependency variants, and custom transformation logic—following the mechanics of Section 5—enables JsShaker to further optimize applications using these libraries. For example, the opt-in React-awareness evaluated in Section 6.1.1 can safely prune stable dependencies from `React.useEffect` calls [Lee et al. 2025].

Generalization to Other Languages. While JsShaker targets JavaScript, its key design, *dependency-aware abstract values*, readily generalizes to other high-level languages, achieving the precision of abstract interpretation with the build-time efficiency of the single-round optimization approach.

8 Related Work

8.1 Abstract Interpretation for JavaScript

Foundational frameworks laid the groundwork for JavaScript static analysis. Jensen et al. [2009] and Kashyap et al. [2014] pioneered the domain, followed by robust infrastructures such as SAFE [Lee et al. 2012; Park et al. 2017b], WALA [Chakraborty et al. 2022], and FAST [Kang et al. 2023].

Precise object property modeling remains a core challenge. Heidegger and Thiemann [2010] introduced recency types for initialization, while Sridharan et al. [2012] addressed correlation tracking for dynamic properties. Cox et al. [2014] refined open object iteration, followed by Park et al. [2017a]'s singleton abstraction to fix non-monotonicity. Recently, Laursen et al. [2024] proposed approximate interpretation to reduce unsoundness.

Distinct from these structural abstractions, Nicolay et al. [2017] focused on behavioral semantics, inferring function purity. To improve scalability and analysis speed, Dewey et al. [2015] and Ko et al. [2015] introduced parallel and tunable frameworks. Park et al. [2018] later applied loop sensitivity

for large libraries. Most recently, Yang et al. [2025] proposed interprocedural weak topological ordering to efficiently decompose recursions and optimize fixpoint computations.

8.2 Single-Round Analysis and Optimization

Traditional compilers suffer from the phase ordering problem, where sequential passes miss cascading optimization opportunities. Early monolithic super-analyses addressed this by fusing optimizations. Wegman and Zadeck [1991] mutually reinforced constant discovery and dead code elimination within a single evaluation. Click and Cooper [1995] combined constant propagation, global value numbering, and dead code elimination into an optimistic fixpoint computation.

To overcome the poor modularity of monolithic designs, Lerner et al. [2002] proposed an implicit communication framework where independent modules interact via tentative graph mutations, committing structural changes only upon reaching a globally sound fixpoint.

Recently, Lemerre [2023] modeled SSA construction as an abstract interpretation, laying the foundation for single-round integration. Lesbre and Lemerre [2024] generalized this by formalizing compiler passes as functor domains acting on free algebras. Consequently, analysis and transformation occur strictly simultaneously: the computed algebraic state intrinsically constitutes the transformed program graph.

8.3 Property Mangling

Some minifiers, such as Terser [Maiwald et al. 2026], UglifyJS [Bazon 2010], and esbuild [Wallace 2020], support opt-in property mangling that relies on lexical heuristics and manual configuration, which scales poorly and risks breaking dynamic accesses at runtime. Conversely, Google Closure Compiler [Google 2026] applies aggressive whole-program mangling, but imposes a prohibitive developer burden by requiring exhaustive external declarations and JSDoc annotations to prevent breakage. While ad-hoc type inference models [Feldthaus and Møller 2013] compute heuristic equivalence groups for property renaming, their tolerance for theoretical unsoundness restricts them to interactive refactoring rather than automated compilation. Alternatively, a VS Code build script [Bierner 2023] leverages TypeScript's refactoring capabilities to mangle properties, but it is restricted to TypeScript codebases and suffers from the unsoundness of TypeScript's type system.

9 Conclusion

This paper presents JsShaker, a framework that reconciles the fundamental trade-off between precision and efficiency in JavaScript optimization. By introducing dependency-aware abstract values into abstract interpretation, code liveness is determined as an intrinsic part of value inference. This enables deep semantic analysis and aggressive code size optimization without the prohibitive cost of iterative analysis–transformation cycles. Furthermore, this architecture allows for modular extensions, ranging from control flow simplification and constant folding to safe property mangling. Empirical results on real-world libraries validate the practicality of this approach: JsShaker achieves an average code size reduction of 27.6% (peaking at 66.7%) with only a 14% increase in build time compared to the evaluated tools, with correctness rigorously verified by the Test262 conformance suite. These findings highlight the potential for semantics-aware optimization to become a standard, scalable component of next-generation Web toolchains.

Data-Availability Statement

An anonymized preliminary version of our artifact, including the source code and benchmark datasets, is available at https://anonymous.4open.science/r/jsshaker-benchmark-903E. Currently, running the artifact requires either a native Linux environment or Docker. We will submit the complete artifact for Artifact Evaluation if the paper is accepted.

References

Mihai Bazon. 2010. UglifyJS — JavaScript Parser, Compressor, Minifier Written in JS. Retrieved February 24, 2026 from https://lisperator.net/uglifyjs/

Matt Bierner. 2023. Shrinking VS Code with Name Mangling. Retrieved February 24, 2026 from https://code.visualstudio.com/blogs/2023/07/20/mangling-vscode

Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, et al. 2022. Automatic root cause quantification for missing edges in JavaScript call graphs. In *Proceedings of the 36th European Conference on Object-Oriented Programming*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 3:1–3:28. doi:10.4230/LIPIcs.ECOOP.2022.3

Moumena Chaqfeh, Russell Coke, Jacinta Hu, et al. 2022a. JSAnalyzer: A Web Developer Tool for Simplifying Mobile Web Pages through Non-critical JavaScript Elimination. *ACM Trans. Web* 16, 4, Article 17 (Nov. 2022), 31 pages. doi:10.1145/3550358

Moumena Chaqfeh, Muhammad Haseeb, Waleed Hashmi, et al. 2022b. To Block or Not to Block: Accelerating Mobile Web Pages On-The-Fly Through JavaScript Classification. In *Proceedings of the 22nd International Conference on Information and Communication Technologies and Development*. ACM, New York, NY, USA, 1–12. doi:10.1145/3572334.3572397

Moumena Chaqfeh, Yasir Zaki, Jacinta Hu, et al. 2020. JSCleaner: De-Cluttering Mobile Webpages Through JavaScript Cleanup. In *Proceedings of the 29th International World Wide Web Conference*. ACM, New York, NY, USA, 763–773. doi:10.1145/3366423.3380157

Boshen Chen et al. 2026. *The JavaScript Oxidation Compiler*. https://oxc.rs/ Originally released in 2023.

Chrome Team. 2023. Why does speed matter? Retrieved February 24, 2026 from https://web.dev/learn/performance/why-speed-matters

Cliff Click and Keith D. Cooper. 1995. Combining Analyses, Combining Optimizations. *ACM Trans. Program. Lang. Syst.* 17, 2 (March 1995), 181–196. doi:10.1145/201059.201061

Cloudflare. 2025. Limits · Cloudflare Workers docs. Retrieved February 24, 2026 from https://developers.cloudflare.com/workers/platform/limits/#worker-size

Jeremy Condit, Juan Chen, Chris Hawblitzel, et al. 2008. Type-Preserving Compilation for Large-Scale Optimizing Object-Oriented Compilers. In *Proceedings of the 29th ACM Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 183–192.

Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 238–252. doi:10.1145/512950.512973

Arlen Cox, Bor-Yuh Evan Chang, and Xavier Rival. 2014. Automatic analysis of open objects in dynamic language programs. In *Proceedings of the 21st International Symposium on Static Analysis*. Springer, Cham, 134–150. doi:10.1007/978-3-319-10936-7_9

Devographics. 2025. State of JS 2025: Build Tools. Retrieved February 24, 2026 from https://2025.stateofjs.com/en-US/libraries/build-tools/

Kyle Dewey, Vineeth Kashyap, and Ben Hardekopf. 2015. A parallel abstract interpreter for JavaScript. In *Proceedings of the 13th IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE, Washington, DC, USA, 34–45. doi:10.1109/CGO.2015.7054186

Ecma International. 2025. *ECMAScript® 2025 Language Specification*. Ecma International. Retrieved February 24, 2026 from https://www.ecma-international.org/publications-and-standards/standards/ecma-262/

Ecma TC39. 2026. Test262: Official ECMAScript Conformance Test Suite. Retrieved February 24, 2026 from https://github.com/tc39/test262

Electron. 2013. Electron. Retrieved February 24, 2026 from https://www.electronjs.org/

engine262 Contributors. 2026. engine262: An implementation of ECMAScript in JavaScript. Retrieved February 24, 2026 from https://github.com/engine262/engine262

Facebook, Inc. 2017. Prepack · Partial Evaluator for JavaScript. Retrieved February 24, 2026 from https://prepack.io/

Fábio de A. Farzat, Márcio de O. Barros, and Guilherme H. Travassos. 2021. Evolving JavaScript Code to Reduce Load Time. *IEEE Transactions on Software Engineering* 47, 8 (Aug. 2021), 1544–1558. doi:10.1109/TSE.2019.2928293

Asger Feldthaus and Anders Møller. 2013. Semi-automatic rename refactoring for JavaScript. In *Proceedings of the 28th International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, USA, 323–338. doi:10.1145/2509136.2509520

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349. doi:10.1145/24039.24041

GitHub. 2025. Octoverse: A new developer joins GitHub every second as AI leads TypeScript to #1. Retrieved February 24, 2026 from https://github.blog/news-insights/octovers/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/

Google. 2019. Business growth through mobile site speed - Think with Google. Retrieved February 24, 2026 from https://business.google.com/in/think/marketing-strategies/mobile-site-speed-importance/

Google. 2026. *Closure Compiler*. https://github.com/google/closure-compiler Originally released in 2009.

David Grove, Greg DeFouw, Jeffrey Dean, et al. 1997. Call Graph Construction in Object-Oriented Languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, USA, 108–124. doi:10.1145/263698.264352

Rich Harris. 2015. Tree-shaking versus dead code elimination. Retrieved February 24, 2026 from https://medium.com/@Rich_Harris/tree-shaking-versus-dead-code-elimination-d3765df85c80

Rich Harris et al. 2026. *Rollup*. https://rollupjs.org/ Originally released in 2015.

Phillip Heidegger and Peter Thiemann. 2010. Recency types for analyzing scripting languages. In *Proceedings of the 24th European Conference on Object-Oriented Programming*. Springer, Berlin, Heidelberg, 200–224. doi:10.1007/978-3-642-14107-2_10

Venter Herman. 2020. Is Prepack Dead? · Issue #2639 · facebookarchive/prepack. Retrieved February 24, 2026 from https://github.com/facebookarchive/prepack/issues/2639#issuecomment-585997029

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis*, Jens Palsberg and Zhendong Su (Eds.). Springer, Berlin, Heidelberg, 238–255. doi:10.1007/978-3-642-03237-0_17

JerryScript. 2015. JerryScript: A lightweight JavaScript engine for Internet of Things. Retrieved February 24, 2026 from https://jerryscript.net/

Dongyoon Kang et al. 2019. swc: Rust-Based Platform for the Web. Retrieved February 24, 2026 from https://swc.rs/

Mingqing Kang, Yichao Xu, Song Li, Rigel Gjomemo, et al. 2023. Scaling JavaScript Abstract Interpretation to Detect and Exploit Node.js Taint-style Vulnerability. In *Proceedings of the 44th 2023 IEEE Symposium on Security and Privacy*. IEEE, San Francisco, CA, USA, 1059–1076. doi:10.1109/SP46215.2023.10179352

Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, et al. 2014. JSAI: A Static Analysis Platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, Edinburgh, United Kingdom, 121–132. doi:10.1145/2635868.2635904

Yoonseok Ko, Hongki Lee, Julian Dolby, et al. 2015. Practically tunable static analysis framework for large-scale JavaScript applications. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, Lincoln, NE, USA, 541–551. doi:10.1109/ASE.2015.60

Igibek Koishybayev and Alexandros Kapravelos. 2020. Mininode: Reducing the Attack Surface of Node.js Applications. In *Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses*. USENIX Association, San Sebastian, 121–134. https://www.usenix.org/conference/raid2020/presentation/koishybayev

Tobias Koppers et al. 2017. Webpack 2.0.0 Release. Retrieved March 9, 2026 from https://github.com/webpack/webpack/releases/tag/v2.0.0

Jesutofunmi Kupoluyi, Moumena Chaqfeh, Matteo Varvello, et al. 2022. Muzeel: assessing the impact of JavaScript dead code elimination on mobile web performance. In *Proceedings of the 22nd ACM Internet Measurement Conference*. ACM, New York, NY, USA, 335–348. doi:10.1145/3517745.3561427

Mathias Rud Laursen, Wenyuan Xu, and Anders Møller. 2024. Reducing Static Analysis Unsoundness with Approximate Interpretation. *Proc. ACM Program. Lang.* 8, PLDI (June 2024), 1165–1188. doi:10.1145/3656424

Hongki Lee, Soonho Won, Joonho Han, et al. 2012. SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript. In *Proceedings of the 2012 Future of Software Engineering*. IEEE, Piscataway, NJ, USA, 125–140. doi:10.11045/fose.2012.9

Jay Lee, Joongwon Ahn, and Kwangkeun Yi. 2025. React-tRace: A Semantics for Understanding React Hooks: An Operational Semantics and a Visualizer for Clarifying React Hooks. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 289 (Oct. 2025), 28 pages. doi:10.1145/3763067

Matthieu Lemerre. 2023. SSA Translation Is an Abstract Interpretation. *Proc. ACM Program. Lang.* 7, POPL, Article 65 (Jan. 2023), 30 pages. doi:10.1145/3571258

Sorin Lerner, David Grove, and Craig Chambers. 2002. Composing Dataflow Analyses and Transformations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 270–282. doi:10.1145/503272.503298

Dorian Lesbre and Matthieu Lemerre. 2024. Compiling with Abstract Interpretation. *Proc. ACM Program. Lang.* 8, PLDI, Article 162 (June 2024), 26 pages. doi:10.1145/3656392

Yuxin Liu, Deepika Tiwari, Cristian Bogdan, et al. 2025. Detecting and removing bloated dependencies in CommonJS packages. *Journal of Systems and Software* 230 (2025), 112509. doi:10.1016/j.jss.2025.112509

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, et al. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. doi:10.1145/2644805

Yinan Long et al. 2026. *NAPI-RS*. https://napi.rs/ Originally released in 2020.

Walter Lucas, Rafael Nunes, Rodrigo Bonifácio, et al. 2025. Understanding the Adoption of Modern Javascript Features: An Empirical Study on Open-Source Systems. *Empirical Software Engineering* 30, 4 (May 2025), 107. doi:10.1007/s10664-025-10663-9

Fabian Maiwald et al. 2026. *Terser*. https://terser.org/ Originally released in 2018.

Ivano Malavolta, Kishan Nirghin, Gian Luca Scoccia, et al. 2023. JavaScript Dead Code Identification, Elimination, and Empirical Assessment. *IEEE Transactions on Software Engineering* 49, 7 (July 2023), 3692–3714. doi:10.1109/TSE.2023.3267848

MDN Contributors. 2025. Primitive - Glossary | MDN. Retrieved February 24, 2026 from https://developer.mozilla.org/en-US/docs/Glossary/Primitive

Jens Nicolay, Quentin Stiévenart, Wolfgang De Meuter, et al. 2017. Purity Analysis for JavaScript through Abstract Interpretation. *Journal of Software: Evolution and Process* 29, 12 (2017), e1889. doi:10.1002/smr.1889

Node.js. 2009. Node.js. Retrieved February 24, 2026 from https://nodejs.org/

npm, Inc. 2026. npm | Home. Retrieved February 24, 2026 from https://www.npmjs.com/

Ayush Pandey, Matteo Varvello, Syed Ishtiaque Ahmed, et al. 2025. Maml: Towards a faster web in developing regions. In *Proceedings of the 34th ACM Web Conference*. ACM, New York, NY, USA, 727–739. doi:10.1145/3696410.3714584

Changhee Park, Hongki Lee, and Sukyoung Ryu. 2018. Static analysis of JavaScript libraries in a scalable and precise way using loop sensitivity. *Software: Practice and Experience* 48, 4 (2018), 911–944. doi:10.1002/spe.2552

Jihyeok Park, Xavier Rival, and Sukyoung Ryu. 2017a. Revisiting recency abstraction for JavaScript: towards an intuitive, compositional, and efficient heap abstraction. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. Association for Computing Machinery, New York, NY, USA, 1–6. doi:10.1145/3088515.3088516

Jihyeok Park, Yeonhee Ryou, Joonyoung Park, et al. 2017b. Analysis of JavaScript Web Applications Using SAFE 2.0. In *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE, Piscataway, NJ, USA, 59–62. doi:10.1109/ICSE-C.2017.4

React Native. 2015. React Native. Retrieved February 24, 2026 from https://reactnative.dev/

Sonatype. 2024. *10th Annual State of the Software Supply Chain Report*. Technical Report. Sonatype. Retrieved February 24, 2026 from https://www.sonatype.com/hubfs/SSCR-2024/SSCR_2024-FINAL-10-10-24.pdf

Manu Sridharan, Julian Dolby, Satish Chandra, et al. 2012. Correlation tracking for points-to analysis of JavaScript. In *Proceedings of the 26th European Conference on Object-Oriented Programming*. Springer, Berlin, Heidelberg, 435–458. doi:10.1007/978-3-642-31057-7_20

Kwangwon Sun and Sukyoung Ryu. 2018. Analysis of JavaScript Programs: Challenges and Research Trends. *Comput. Surveys* 50, 4 (July 2018), 1–34. doi:10.1145/3106741

Alexi Turcotte, Ellen Arteca, Ashish Mishra, et al. 2022. Stubbifier: debloating dynamic server-side JavaScript applications. *Empirical Software Engineering* 27, 7 (12 2022), 1–36. doi:10.1007/s10664-022-10195-6

Tomoharu Ugawa, Hideya Iwasaki, and Takafumi Kataoka. 2019. eJSTK: Building JavaScript virtual machines with customized datatypes for embedded systems. *Journal of Computer Languages* 51 (2019), 261–279.

VoidZero Inc. 2026. *Rolldown | Rust Bundler for JavaScript*. https://rolldown.rs/ Originally released in 2023.

Evan Wallace. 2020. esbuild - An Extremely Fast Bundler for the Web. Retrieved February 24, 2026 from https://esbuild.github.io/

Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems* 13, 2 (apr 1991), 181–210. doi:10.1145/103135.103136

Jiawei Yang, Xiao Cheng, Bor-Yuh Evan Chang, et al. 2025. Taming and Dissecting Recursions Through Interprocedural Weak Topological Ordering. In *Proceedings of the 39th European Conference on Object-Oriented Programming*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 34:1–34:25. doi:10.4230/LIPIcs.ECOOP.2025.34