Detecting JavaScript Libraries on the Web using Property-Order Graphs

SUNGMIN PARK, Korea University, South Korea
SEOJIN KIM, North Carolina State University, USA
BEN SWIERZY, Fraunhofer FKIE, Germany and University of Bonn, Germany
JIHYEOK PARK*, Korea University, South Korea

Detecting front-end JavaScript libraries is essential for website profiling, vulnerability detection, and dependency management. However, bundlers transpile code in various ways, altering its original structure and complicating detection. State-of-the-art techniques detect libraries at runtime by matching property patterns exposed through the global object. They therefore miss libraries inaccessible from the global object, and identify versions only when version labels are exposed or property-level differences are large. More recent approaches inspect the bundled source, but either target a single bundler or require the target library as input. We present Debun, a scalable technique for detecting JavaScript libraries and their versions using function-level fingerprints. Our key insight is that bundlers preserve property names and the execution order of property operations. We introduce the *property-order graph* (POG), which captures this order within a function body, and make it robust to interprocedural transformations such as function inlining through preprocessing. To detect partially imported libraries caused by tree shaking, we propose a dependency-aware scoring function. Evaluations on 68 high-traffic websites with 78 libraries show that Debun outperforms five state-of-the-art tools, achieving a 94.64% F1-score in library detection and identifying 88 of the 98 ground-truth versions at the patch level.

CCS Concepts: • **Security and privacy → Software reverse engineering**; • **Software and its engineering → Automated static analysis**; *Software libraries and repositories*.

Additional Key Words and Phrases: JavaScript, library detection, version detection, bundlers, static analysis, function fingerprinting

1 Introduction

Detecting front-end JavaScript libraries in web applications is essential for diverse tasks. Platforms like W3Techs leverage this data to track JavaScript library usage, and security researchers use it to identify well-known vulnerabilities in third-party libraries [12, 29]. Static analyzers also enhance precision and efficiency by incorporating predefined API modeling of detected libraries [5, 24]. According to W3Techs' survey, 98.8% of all websites use JavaScript, and 81.4% use third-party libraries. Such extensive use of libraries in web applications necessitates accurate detection techniques.

*Corresponding author

Authors' Contact Information: Sungmin Park, Korea University, Department of Computer Science and Engineering, Seoul, South Korea, ryan040@korea.ac.kr; Seojin Kim, North Carolina State University, Department of Computer Science, Raleigh, North Carolina, USA, skim253@ncsu.edu; Ben Swierzy, Fraunhofer FKIE, Department of Cyber Security, Bonn, Germany and University of Bonn, Department of Computer Science, Bonn, Germany, swierzy@cs.uni-bonn.de; Jihyeok Park, Korea University, Department of Computer Science and Engineering, Seoul, South Korea, jihyeok_park@korea.ac.kr.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM 1557-7392/2026/11-ART
https://doi.org/10.1145/3862477

Just Accepted

Challenges. In the past, when developers typically included libraries as separate, unmodified script files, simple methods such as matching filenames, URLs, or hashes of the served files were sufficient to identify library usage on the web [13]. However, with the emergence and prevalence of more advanced tooling, detecting libraries in web applications has become challenging because bundlers like Webpack or Rollup transpile the code and obscure its original structure. Developers use bundlers to lower the network cost of loading an application by reducing the size of the transferred files. Bundlers combine multiple JavaScript files into a single or a few output files and apply transformations, such as minification, dead-code elimination, and tree shaking. A recent study reported that 40% of the top 1M websites serve bundles containing third-party libraries [26]. After bundling, the original directory structure is lost, making it hard to distinguish between first-party and third-party code. It also hinders the use of directory structure-based techniques [22, 36] to detect JavaScript libraries in web applications. The transformations applied by bundlers can mangle variable names, compress the code, and even significantly alter the code structure through advanced optimizations. Such complex modifications make it infeasible to naively apply the code clone detection techniques [16, 33, 37] developed for other programming languages to JavaScript library detection.

Limitations of Existing Techniques. The state-of-the-art approach for detecting JavaScript libraries relies on patterns of object properties reachable from the global object to identify libraries at runtime. A popular open-source tool integrated into Chrome Lighthouse, Library Detector For Chrome (LDC) [1], employs this approach but with a manually defined set of detection patterns. Liu and Ziarek [18] introduce PTDetector, a tool that automatically extracts *pTree* data structures, which represent the tree of reachable properties, to detect libraries. PTV [17] further extends PTDetector to identify library versions from the same runtime property patterns. However, these property pattern-based techniques share two key limitations: (1) they cannot detect libraries inaccessible from the global object, and (2) they identify versions only when the exposed property patterns differ, since they never inspect function bodies.

Therefore, it is necessary to consider the code structure of function bodies to accurately detect libraries and their versions. One possible way is to exhaustively transpile libraries with all possible configurations of bundlers and collect fingerprints to detect libraries [3]. This is inefficient because bundlers generate exponentially many different code depending on the configuration. If a bundler supports n boolean options, it generates 2^n fingerprints for each library. More recent approaches examine the source code itself. D-Bundlr [38] classifies the modules recovered from bundled files with a learned model. Although it inspects source code, its learning-based approach targets only JavaScript code bundled by Webpack. Aletheia [31] detects only library versions, not libraries. Given a target library as input, it identifies which versions of that library are bundled through hash-based matching against its npm releases.

Our Approach. To alleviate these limitations, we propose Debun, a scalable tool for JavaScript library and version detection using function-level fingerprints. Debun extracts fingerprints by constructing a *property-order graph* (POG) that captures the execution order of property operations in function bodies. This graph represents **1)** which property *operations* on **2)** which property *names* are executed in **3)** which *order* in a function body, with a consideration of *control flow*. Our key insight is that the property names and execution order of property operations remain after transpilation, although transpilers often change the control flow of the original code.

We explain when control flows are *inconsistent* between the original and transpiled code (§2), and propose how to construct consistent POGs with a *path-sensitive truthiness analysis* (§4) based on abstract interpretation frameworks [6, 7]. Then, we detect libraries and versions by comparing POG-based fingerprints of target applications and libraries. We implement our approach in Debun and evaluate it on 68 high-traffic websites with 78 front-end JavaScript libraries. Our experimental results demonstrate that Debun outperforms five state-of-the-art tools, achieving a 94.64% F1-score in library detection and correctly identifying 88 of the 98 ground-truth versions at the patch level.

Just Accepted

```
1 function arrayEach (array, iteratee){
2   var index = -1,
3       length = array == null ? 0 : array.length;
4   while (++index < length) {
5     if (iteratee(array[index],index,array)===false)
        break;
6   }
7   return array;
8 }
```

(a) Original arrayEach function in Lodash.js v4.17.21.

```
1 // npm-596046b7.0a1ae60586ca5609f0c5.js
2 // in tiktok.com
3 ... = function(r, t) {
4   for (var e = -1,
5        n = null == r ? 0 : r.length;
6        ++e < n && !1 !== t(r[e], e, r)
       ;);
7   return r
8 }
```

(b) Transpiled arrayEach function in the TikTok website.

Fig. 1. The original arrayEach function of Lodash.js v4.17.21 and its transpiled code in the TikTok website.

Contributions. The contributions of this paper are as follows:

- We introduce the *property-order graph* (POG) to capture the execution order of property operations. We further enhance its robustness by introducing a *function inlining* preprocessing pass.
- We propose a *dependency-aware scoring* function that utilizes module-level dependency graphs to accurately detect libraries even when only partial modules remain due to tree-shaking.
- We implement Debun and conduct an extensive evaluation on 68 high-traffic websites. We show that Debun significantly outperforms five state-of-the-art tools in both library and version detection.

This article extends a previously published conference paper [10] along three directions. **First**, we introduce a *function inlining* (§5) preprocessing pass to enhance the robustness of POGs. If a function is called only once, minifiers often inline its body at the call site, which merges the function's POG with that of its caller. For robustness against such transformations, our preprocessing pass identifies functions that are called exactly once and inlines their bodies at the call sites before generating POGs. **Second**, we propose a *dependency-aware library scoring* function (§6.2) to detect libraries and versions more accurately. Bundlers often perform tree shaking optimization, which includes only the modules of a library that are imported by the application. While the previous scoring function matched the fingerprints of the whole library against the target application, the new scoring function considers the dependency module graph of each library to perform a more fine-grained matching against the target application. **Third**, we extensively expand the empirical evaluation of Debun (§7). We additionally compare against three recent state-of-the-art tools: D-Bundlr [38] for library detection, and Aletheia [31] and PTV [17] for version detection. We refine the version-detection metrics with three levels of granularity: major, minor, and patch, to provide a more detailed analysis of version detection performance. We extend the consistency and accuracy ablation with the new function-inlining variant, and add a new ablation study on the dependency-aware library scorer.

2 Background and Motivation

This section shows the limitations of existing techniques in detecting libraries and their versions in web applications, and then presents the insight behind our technique and its challenges using the real-world example of Figure 1.

2.1 Existing Techniques and Their Limitations

There are two main existing approaches to detect libraries or their versions in web applications.

Just Accepted

2.1.1 Property Pattern-based Detection at Runtime. The most common approach relies on patterns of the object properties reachable from the global object at runtime. Library Detector For Chrome (LDC) [1] detects Lodash.js[1] by checking the existence of the property `_` in the global object `win` and its property `_.chain` with the following conditions:

```
typeof (_ = win._) == 'function' && _
typeof (chain = _ && _.chain) == 'function' && chain
```

After detecting the library, LDC determines its version with the version-specific property `_.VERSION`:

```
return { version: _.VERSION || UNKNOWN_VERSION }
```

However, it heavily relies on manually defined patterns, which are moreover not always discriminating: Underscore.js exposes the same `_` and `_.chain` properties as Lodash.js, so the rules above match both.

This limitation is addressed by the *pTree* data structure in the tool PTDetector [18]. A *pTree* represents the tree structure of all reachable properties from the global object at runtime. While the tool automatically detects libraries by comparing the pTrees of the web application and of known libraries, property pattern-based techniques still share two key limitations. First, they cannot detect libraries *internally imported* by bundlers: libraries embedded inside the bundle as *internal modules* registered under bundler-internal IDs or immediately-invoked function expressions (IIFEs), and thus never exposed as properties of the global object. Only libraries *loaded externally* as standalone scripts, typically from a content delivery network (CDN), keep the global exports that these tools rely on. For example, the `arrayEach` function of Lodash.js is imported as an internal module keyed by the bundler-internal ID `83271` in the TikTok website:

```
(...).push([ [3824], { ..., 83271: (e,t,n) => { r.exports = /* Fig. 1b */ }, ... }, ]);
```

Second, property patterns are not sufficient to detect library versions when no version-specific property exists. A version update often changes the internal code structure of a library without changing its property pattern, so neglecting function bodies leaves such versions indistinguishable. For example, compared to jQuery v3.7.0, its patch version v3.7.1 has the same property pattern but different bodies of the `text` and `hover` functions.[2] PTV [17], which extends PTDetector to version detection, therefore cannot tell such versions apart.

2.1.2 Transpiled Code Detection. Therefore, it is necessary to consider the code structure of function bodies to detect libraries and their versions accurately. Most existing code clone detection techniques only consider simple syntactic transformations [11, 14, 41] (e.g., formatting and variable renaming) or control/data flows [43, 45]. However, JavaScript bundlers alter the code structure through advanced optimizations that even change the control and data flow, as Figure 1 illustrates, so these techniques cannot be applied naively.

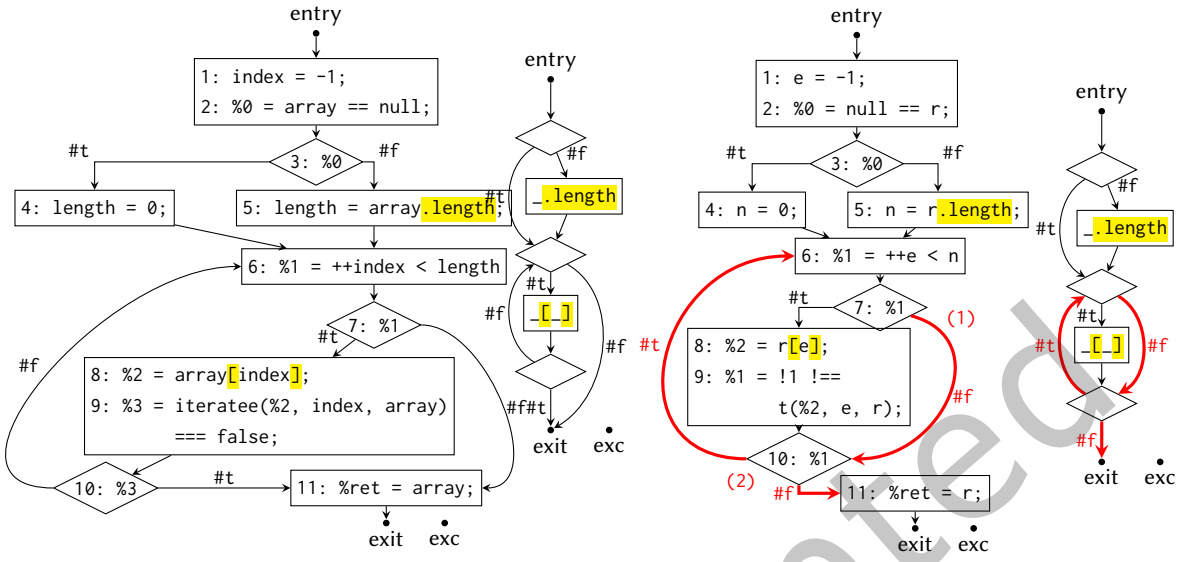
Another approach is to exhaustively transpile library code under all bundler configurations and collect their fingerprints. However, bundlers support tens of configurable options (e.g., Terser alone has 34), so the number of possible configurations grows exponentially (2^{34} for Terser), requiring large storage space. For example, the tool URR [3] suffers from this inefficiency. Even though it uses only 24,576 configurations of Webpack, it requires 2.27GB of storage space for only three libraries. Thus, it is not scalable when detecting a large number of libraries in web applications.

2.2 Our Approach: Property-Order Graphs

Instead, we focus on what bundlers retain in the code after transpilation.

[1] https://lodash.com/
[2] https://github.com/jquery/jquery/compare/3.7.0...3.7.1

Just Accepted



(a) CFG of original code. (b) POG of (a). (c) CFG of bundled code. (d) POG of (c).

Fig. 2. CFGs of original and bundled code in Figure 1 and their corresponding POGs constructed by the basic algorithm.

Key Insight. We observe that property names and the execution order of property operations remain unchanged after bundling. This assumption is reasonable because JavaScript's support for computed property names and getter/setter-based property access makes property operations difficult to resolve statically. In JavaScript, properties can be accessed using computed names; `obj['h'+'i']` reads the property `hi` of the object `obj`. Thus, bundlers must retain property names, since a computed access may resolve to any of them. For example, Figure 1 shows that a property read operation highlighted in yellow with the name `length` is preserved in the transpiled code.

In addition, property reads and writes may invoke user-defined getter and setter functions. Since they may have side effects, bundlers retain their existence and execution orders to maintain the original behavior of the code. For example, two property read operations in the original code (Figure 1a) are preserved with the same execution order in the transpiled code (Figure 1b). Considering the execution order of property operations increases the precision of library detection. For example, the following function in can.js v6.6.3 is a false match without considering the execution order of property operations:

```
// can.js - false alarm without execution order
function (arr) { return arr != null && arr[arr.length - 1]; }
```

This false match arises because the function performs the same two property read operations as `arrayEach`: one with the name `length` and another with the computed name `[_]`. Their execution orders nevertheless differ, since `arrayEach` reads `length` once and then `[_]` once per loop iteration, whereas can.js reads each of the two exactly once. A POG captures this difference because it inherits the loop's back edge from the control flow.

To capture this insight, we introduce the *property-order graph* (POG) that represents the execution order of property operations with their names in function bodies. A POG represents **1)** which property *operations* on **2)** which property *names* are executed in **3)** which *order* in a function body, with a consideration of *control flow*. Concretely, a POG is derived from the control-flow graph (CFG) of a function body by keeping only its property

Just Accepted

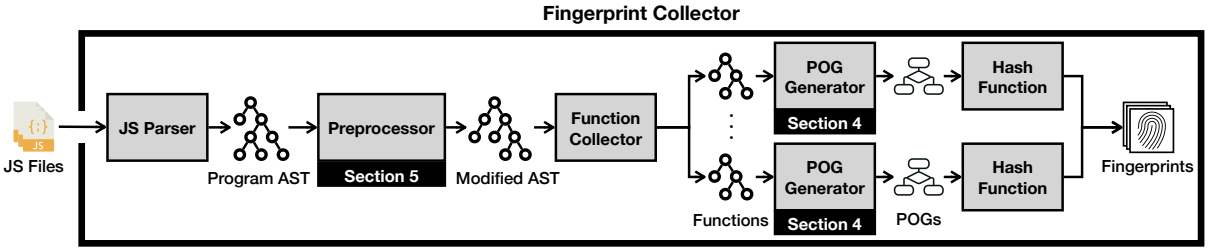


Fig. 3. The fingerprint collector for given JavaScript files.

operations and branches. §4.1 constructs POGs step by step and gives the formal definition (Definition 4.1). For example, Figures 2a and 2c depict CFGs of the original and bundled code in Figure 1. Each rectangle represents a sequence of normal instructions, and each diamond represents a conditional instruction. Their POGs are constructed by filtering property operations from the CFGs as shown in Figures 2b and 2d. In addition, JavaScript transpilers heavily utilize the short-circuit evaluation of logical expressions to reduce code size. To accurately construct POGs, we explicitly represent these logical expressions as branches. For example, the combination of `while`-loop and `if`-statement in Figure 1a is converted into a `for`-loop and a logical AND expression in Figure 1b.

Challenges – Inconsistent Control Flows. Although the original and bundled functions are semantically equivalent, naively applying the above POG construction to their CFGs yields structurally different graphs as shown in Figures 2b and 2d (highlighted in thick red lines). The discrepancy arises because the CFG (Figure 2c) of the bundled code contains (1) an infeasible execution path and (2) a flipped conditional branch. First, a path `7` → `10` → `6` is infeasible. The temporary variable `%1` is always falsy at `10` when the execution path comes from the false branch of `7`. This means that all the execution paths from the false branch of `7` always flow into the false branch of `10`, making the true branch of `10` infeasible. Second, the branches of `10` are flipped compared to the CFG of the original code in Figure 2a, because transpilers often negate conditions and flip branches to minimize the code size. Because such bundler-induced artifacts make naively constructed POGs diverge even for semantically equivalent code and may cause false negatives in library detection, we need a dedicated analysis that normalizes these inconsistencies before POG construction.

3 Overview

Before detailing the construction of POGs, this section gives an overview of Debun: it first builds a database of fingerprints from known library versions, and then matches the fingerprints of a target website against that database. Both sides are processed by the same *fingerprint collector* (§3.1), which turns a set of JavaScript files into function-level fingerprints, and §3.2 describes the two-phase structure built around it.

3.1 Fingerprint Collector

As depicted in Figure 3, the fingerprint collector extracts the fingerprints of all functions in given JavaScript files through five components:

- The **JavaScript Parser** produces an abstract syntax tree (AST) for each script or module file using a parser that supports the latest JavaScript syntax.
- The **Preprocessor** applies a general *function inlining* transformation (§5) to canonicalize minifier-applied function inlining, which would otherwise merge multiple functions into a single fingerprint.
- The **Function Collector** traverses the preprocessed ASTs and collects all functions (declarations, expressions, arrow functions, and methods) along with their bodies.

Just Accepted

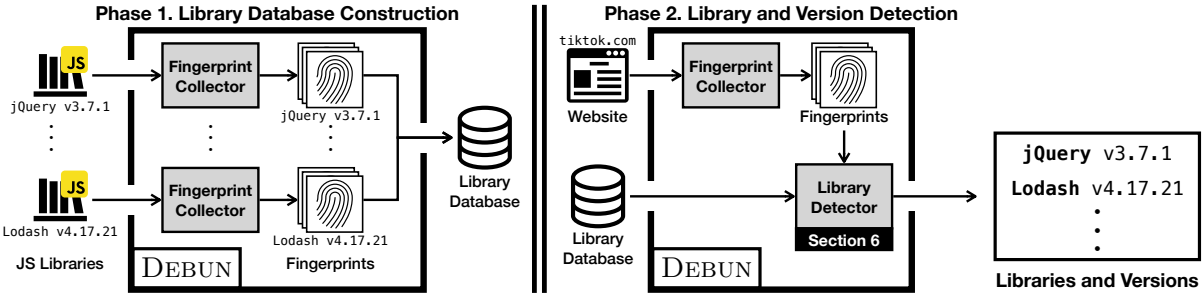


Fig. 4. Overall structure of DEBUN with two phases: 1) library database construction and 2) library (version) detection.

- The **POG Generator** constructs a POG for each collected function to capture the execution order of its property operations, normalizing the control flow that bundlers rewrite (§4).
- The **Hash Function** summarizes the POG of each function into a single fingerprint, yielding one fingerprint per function.

3.2 Overall Structure of DEBUN

Figure 4 depicts the overall structure of DEBUN, which consists of two main phases:

- **Phase 1: Library Database Construction.** DEBUN first applies the fingerprint collector to known libraries to extract function-level fingerprints, and then builds a library database that maps each fingerprint to its corresponding library versions. Because fingerprints that recur across a clear majority of unrelated websites provide little library-specific evidence, we remove such ubiquitous fingerprints from the database, using the frequency threshold reported in §7.1.
- **Phase 2: Library and Version Detection.** DEBUN applies the same fingerprint collector to a target website and compares the collected fingerprints against the database using a *similarity score* to detect libraries and their versions (§6). Scoring a library as a single unit misses the ones that tree shaking includes only partially, so DEBUN scores each library at the granularity of its modules and their dependencies.

The remainder of this paper is organized as follows. We first present the POG construction algorithm (§4), which embodies the key idea of DEBUN, and then the general function inlining preprocessing pass that precedes it in the pipeline (§5). Then, we introduce the library and version detection methods based on similarity scoring (§6). Finally, we evaluate DEBUN on real-world benchmarks (§7), discuss its limitations (§8) and related work (§9), and conclude the paper (§10).

4 Construction of Property-Order Graphs

This section describes how DEBUN constructs POGs. The basic algorithm in §4.1 builds the CFG of each function and keeps only its property operations and branches, yielding a basic POG. However, bundler-induced artifacts create inconsistent control flows between original and bundled code (§2.2), so this basic algorithm produces structurally different POGs for semantically equivalent code. To normalize these inconsistencies, the *path-sensitive truthiness analysis* in §4.2 computes, for each execution path, whether each branch condition is definitely truthy or falsy and whether its truthiness has been flipped by negation. Using only this information, §4.3 normalizes the CFG in three steps: branch flipping, branch bypassing, and path cloning. The CFGs of original and bundled code then converge to a common normal form. Since POGs are extracted from the normalized CFGs, both versions of the code yield the same fingerprint.

Just Accepted

$$
\begin{array}{lll}
\textbf{CFGs} & & \mathcal{G} ::= \{\ell{:}i^*\}\ (\ell_{\text{entry}}, \ell_{\text{exit}}, \ell_{\text{exc}}) \\
\textbf{Instructions} & & i ::= x = e;\ \ell \mid x = e.p;\ \ell \mid e.p = e;\ \ell \mid x = e[e];\ \ell \mid e[e] = e;\ \ell \mid \texttt{if}\ (e)\ \ell\ \texttt{else}\ \ell \mid \ldots \\
\textbf{Expressions} & \mathbb{E} \ni & e ::= c \mid x \mid \texttt{!}e \mid \texttt{\{\}} \mid e(e^*) \mid e == e \mid e === e \mid e < e \mid e > e \mid e \oplus e \mid \ominus e \mid \ldots \\
\textbf{Constants} & & c ::= b \mid n \mid z \mid s \mid \texttt{undefined} \mid \texttt{null}
\end{array}
$$

Variables	$x \in \mathbb{X}$	**Numbers**	$n \in \mathbb{N}$	**Strings**	$s \in \mathbb{S}$	**Labels**	$\ell \in \mathbb{L}$
Properties	$p \in \mathbb{P}$	**BigInts**	$z \in \mathbb{Z}$	**Booleans**	$b \in \mathbb{B} = \{\texttt{\#t}, \texttt{\#f}\}$		

Fig. 5. Control-flow graphs of JavaScript functions.

4.1 Basic Construction Algorithm

A basic construction algorithm of POGs has two steps: 1) construct control-flow graphs (CFGs) of given functions, and 2) keep only their property operations and conditional instructions.

4.1.1 CFG Construction. We use a standard algorithm to construct CFGs defined in Figure 5 where A^* denotes a sequence of A. All instructions in a CFG are labeled with $\mathbb{L}$, and three special labels ℓ_{entry}, ℓ_{exit}, and ℓ_{exc} represent the entry, exit, and exceptional exit points of the function, respectively. An instruction i is either:

- A *normal* instruction with its next label.
- A *conditional* instruction with two labels for branches.

Each JavaScript loop statement (e.g., `while` and `for`) is converted into a conditional instruction whose true branch label points to its loop body, and the false branch label points to the instruction following the loop. Additionally, as explained in §2.2, short-circuit expressions are used as branches. Thus, we convert the following expressions into conditional instructions with temporary variables to mimic their evaluation semantics:

```
%0 = e1; if(%0) { %0 = e2; } else {          } // e1 && e2
%0 = e1; if(%0) {          } else { %0 = e2; } // e1 || e2
%0 = e1; if(%0) { %0 = e2; } else { %0 = e3; } // e1 ? e2 : e3
```

A normal instruction is either: 1) an assignment instruction, 2) a property read/write operation, or 3) a computed property read/write operation. A `return` or a `throw` statement is converted into an assignment instruction to a special variable `%ret` or `%exc`, respectively, and its next label is the exit label or the enclosing `catch` block if it exists, or the exceptional exit label otherwise. Since we construct CFGs to construct POGs, we convert all property read/write operations into normal instructions with temporary variables (e.g., `%0`, `%1`, ...) to explicitly capture the execution order of property operations. For example, `x.p + y.q` is converted into two instructions `%0 = x.p; %1 = y.q;` and an expression `%0 + %1` to represent that `x.p` is executed before executing `y.q`. The inequality and comparison operators that the grammar omits are converted into the remaining ones with negation operators (e.g., `x != y` to `!(x == y)`). A constant is a boolean $b \in \mathbb{B}$, number $n \in \mathbb{N}$, bigint $z \in \mathbb{Z}$, string $s \in \mathbb{S}$, `undefined`, or `null`. The grammar includes all constant categories, not only booleans, because the truthiness analysis must evaluate constants of every category (§4.2.1).

4.1.2 POG Construction. A POG has the same form as a CFG, that is, a set of labeled nodes with entry, exit, and exceptional exit labels. Its nodes, however, are only property operations and conditional instructions.

Definition 4.1 (Property-Order Graph)**.** The *property-order graph* of a function is the directed graph defined in Figure 6, obtained from the function's CFG by keeping only 1) *normal* nodes, which retain the abstracted property

Just Accepted

$$\textbf{POGs} \quad \mathcal{G} ::= \{\ell : w^*\}\ (\ell_{\mathsf{entry}}, \ell_{\mathsf{exit}}, \ell_{\mathsf{exc}})$$
$$\textbf{Nodes} \quad w ::= _.p;\ \ell \mid _.p = _;\ \ell \mid _[_];\ \ell \mid _[_] = _;\ \ell \mid \texttt{if}\ (_)\ \ell\ \texttt{else}\ \ell$$

Fig. 6. Property-order graphs of JavaScript functions.

$$\textbf{Path-Sensitive Results} \quad \hat{\xi} \in \widehat{\Xi} = (\mathbb{L} \times \widehat{\Pi}) \to \widehat{\Sigma}$$
$$\textbf{Abstract Paths} \quad \hat{\pi} \in \widehat{\Pi} = (\mathbb{L} \times \mathbb{B}) \uplus \{\bot\}$$
$$\textbf{Abstract States} \quad \hat{\sigma} \in \widehat{\Sigma} = \mathbb{X} \to \widehat{\mathbb{V}}$$

$$\textbf{Abstract Values} \quad \hat{v} \in \widehat{\mathbb{V}} = \left\{ \begin{array}{ccccccc} & & & \top & & & \\ & & \diagup & \diagup & \diagdown & & \\ \mathsf{t} & & & \mathsf{f} & & F & N \\ & & \diagdown & & \diagup & & \\ & & & \bot & & & \end{array} \right\}$$

Fig. 7. Abstract domains for path-sensitive truthiness analysis, tracking the truthiness (t/f) or flip status (*F*/*N*) of each variable along each execution path.

reads and writes with their names, and 2) *conditional* nodes with two outgoing branch edges; its directed edges represent the execution order between the nodes inherited from the control flow of the CFG.

We derive a POG from a CFG by filtering only four kinds of property operations and conditional instructions. We retain only the property names for property read/write operations; $x = e.p$ and $e_1.p = e_2$ are converted into $_.p$ and $_.p = _$, respectively. For computed property operations, we retain only whether they are read or write operations: $x = e_1[e_2]$ and $e_1[e_2] = e_3$ into $_[_]$ and $_[_] = _$, respectively. As an exception, when the computed property names are constant strings or numeric literals, we normalize them to the same property name; for instance, e`["name"]` becomes `_.name` and both e`[3]` and e`["3"]` become `_.3`. For conditional instructions, we keep only their labels; `if` (e) ℓ_1 `else` ℓ_2 into `if` $(_)$ ℓ_1 `else` ℓ_2. Removing the other instructions splices the control flow: the predecessors of a removed instruction are connected directly to its successor, so an edge of a POG represents reachability through the removed instructions. Figures 2a and 2c depict CFGs of the original and bundled code in Figure 1, and Figures 2b and 2d depict POGs derived from the CFGs.

We also cover modern JavaScript short-circuit syntax (e.g., the nullish coalescing operator `??` and the optional chaining operator `?.`) by likewise treating them as conditional branches. Property writes are often transpiled to object literals, so we consider properties in object literals as property writes (e.g., `{a: b}` is a property write of `a`). Since some expressions (`o.p++`) often contain both reads and writes, we treat them as both with regard to their correct execution order. We omit property operations on built-ins (e.g., `Math`, `Object`) and function-local pseudo-objects (e.g., `arguments[2]`), since they are free of observable side effects in practice and transpilers therefore reorder or eliminate them freely. We also ignore property operations inside inner functions when generating the POG of an outer function, as those operations are executed only when the inner function is explicitly called; each inner function yields its own POG. Our inlining pass introduces one exception to this rule, which we explain together with the transformation itself (§5.2).

4.2 Path-Sensitive Truthiness Analysis

We introduce a *path-sensitive truthiness analysis* based on abstract interpretation [6, 7] to refine control-flows in CFGs for constructing consistent POGs. The only consumer of the analysis result is the flow refinement in §4.3, which needs exactly two pieces of information at each branch: whether its condition is definitely truthy or falsy on a given execution path, and whether the truthiness of the condition has been flipped by negation operators along the way. To compute this information, the analysis tracks the truthiness of each variable along each execution path partitioned by the latest conditional instruction.

Just Accepted

$$\boxed{\widehat{[\![i]\!]}_i : (\mathbb{L} \times \widehat{\Pi} \times \widehat{\Sigma}) \to \widehat{\Xi}}$$

$$\textsc{Cond}\ \frac{\hat{\xi}_1 = \mathsf{refine}(\ell, \hat{\pi}, \hat{\sigma})(\ell_1, e, \#\mathtt{t}) \qquad \hat{\xi}_2 = \mathsf{refine}(\ell, \hat{\pi}, \hat{\sigma})(\ell_2, e, \#\mathtt{f})}{\widehat{[\![\mathtt{if}\ (e)\ \ell_1\ \mathtt{else}\ \ell_2]\!]}_i(\ell, \hat{\pi}, \hat{\sigma}) = \hat{\xi}_1 \sqcup \hat{\xi}_2} \qquad \textsc{Assign}\ \frac{\widehat{[\![e]\!]}_e(\hat{\sigma}) = (\hat{v}, \hat{\sigma}')}{\widehat{[\![x = e;\ \ell']\!]}_i(\ell, \hat{\pi}, \hat{\sigma}) = \{(\ell', \hat{\pi}) \mapsto \hat{\sigma}'[x \mapsto \hat{v}]\}}$$

$$\textsc{Read}\ \frac{}{\widehat{[\![x = e.p;\ \ell']\!]}_i(\ell, \hat{\pi}, \hat{\sigma}) = \{(\ell', \hat{\pi}) \mapsto \hat{\sigma}_t\}} \qquad \textsc{Write}\ \frac{}{\widehat{[\![e_1.p = e_2;\ \ell']\!]}_i(\ell, \hat{\pi}, \hat{\sigma}) = \{(\ell', \hat{\pi}) \mapsto \hat{\sigma}_t\}}$$

$$\textsc{ComputedRead}\ \frac{}{\widehat{[\![x = e_1[e_2];\ \ell']\!]}_i(\ell, \hat{\pi}, \hat{\sigma}) = \{(\ell', \hat{\pi}) \mapsto \hat{\sigma}_t\}} \qquad \textsc{ComputedWrite}\ \frac{}{\widehat{[\![e_1[e_2] = e_3;\ \ell']\!]}_i(\ell, \hat{\pi}, \hat{\sigma}) = \{(\ell', \hat{\pi}) \mapsto \hat{\sigma}_t\}}$$

$$\boxed{\widehat{[\![e]\!]}_e : \widehat{\Sigma} \to (\widehat{\mathbb{V}} \times \widehat{\Sigma})}$$

$$\widehat{[\![c]\!]}_e(\hat{\sigma}) = \begin{cases} (\mathrm{f}, \hat{\sigma}) & \text{if } c \in \{\mathtt{false}, \mathtt{undefined}, \mathtt{null}, \mathtt{+0}, \mathtt{-0}, \mathtt{NaN}, \mathtt{0n}, \mathtt{""}\} \\ (\mathrm{t}, \hat{\sigma}) & \text{otherwise} \end{cases} \qquad \widehat{[\![x]\!]}_e(\hat{\sigma}) = (\hat{\sigma}(x), \hat{\sigma})$$

$$\widehat{[\![!e]\!]}_e(\hat{\sigma}) = \begin{cases} (\mathrm{t}, \hat{\sigma}') & \text{if } \widehat{[\![e]\!]}_e(\hat{\sigma}) = (\mathrm{f}, \hat{\sigma}') \\ (\mathrm{f}, \hat{\sigma}') & \text{if } \widehat{[\![e]\!]}_e(\hat{\sigma}) = (\mathrm{t}, \hat{\sigma}') \\ (F, \hat{\sigma}') & \text{if } \widehat{[\![e]\!]}_e(\hat{\sigma}) = (N, \hat{\sigma}') \\ (N, \hat{\sigma}') & \text{if } \widehat{[\![e]\!]}_e(\hat{\sigma}) = (F, \hat{\sigma}') \\ \widehat{[\![e]\!]}_e & \text{otherwise} \end{cases}$$

$$\widehat{[\![e_1 == e_2]\!]}_e(\hat{\sigma}) = (N, \hat{\sigma}) \qquad \widehat{[\![\{\}]\!]}_e(\hat{\sigma}) = (\mathrm{t}, \hat{\sigma})$$

$$\widehat{[\![e_1 === e_2]\!]}_e(\hat{\sigma}) = (N, \hat{\sigma}) \qquad \widehat{[\![e(e_k^*)]\!]}_e(\hat{\sigma}) = (N, \hat{\sigma}_t)$$

$$\widehat{[\![e_1 < e_2]\!]}_e(\hat{\sigma}) = (N, \hat{\sigma}) \qquad \widehat{[\![e_1 \oplus e_2]\!]}_e(\hat{\sigma}) = (N, \hat{\sigma})$$

$$\widehat{[\![e_1 > e_2]\!]}_e(\hat{\sigma}) = (N, \hat{\sigma}) \qquad \widehat{[\![\ominus e]\!]}_e(\hat{\sigma}) = (N, \hat{\sigma})$$

Fig. 8. The abstract semantics for instructions $\widehat{[\![-]\!]}_i$ and expressions $\widehat{[\![-]\!]}_e$ in the path-sensitive truthiness analysis, defining how the per-path truthiness and flip status of variables are propagated through each instruction.

4.2.1 Abstract Domains. Figure 7 shows the abstract domains for path-sensitive truthiness analysis. The domains formalize *what* the analysis tracks: for each execution path and each variable, either a definite truthiness or a flip status. A path-sensitive result $\hat{\xi} \in \widehat{\Xi}$ is a mapping from pairs of labels and abstract paths to abstract states. An abstract path $\hat{\pi} \in \widehat{\Pi}$ is either 1) a pair of a label and a boolean value (ℓ, b) that represents the true or false branch of the latest conditional instruction labeled by ℓ or 2) $\bot$ for no conditional instruction along the execution path. Note that an abstract path retains only the *latest* conditional instruction and discards earlier branch history; it is thus a bounded, depth-one form of path sensitivity. This bound supplies the two pieces of information that the flow refinement needs, whereas tracking full branch histories would grow the number of abstract paths exponentially. As a trade-off, correlations between distant conditional instructions are not captured and may leave some flows unrefined. An abstract state $\hat{\sigma} \in \widehat{\Sigma}$ is a mapping from variables to abstract values. An abstract value $\hat{v} \in \widehat{\mathbb{V}}$ denotes either 1) the truthiness of a variable (t or f) or 2) whether its truthiness is flipped by negation operators (F or N). The partial order ($\sqsubseteq$) and join ($\sqcup$) of $\widehat{\mathbb{V}}$ are defined by the lattice shown as a Hasse diagram in Figure 7. All JavaScript values are either *truthy* or *falsy* according to the **ToBoolean** algorithm in the language semantics[3]; `false`, `undefined`, `null`, `+0`, `-0`, `NaN`, `0n`, and `""` are falsy values, and other primitive values and objects are truthy values. The abstract values t and f mean that only truthy and falsy values are possible, respectively. On the other hand, the abstract values F and N denote whether the truthiness is flipped or not, respectively, through a sequence of variable assignments without any side effects.

[3]https://tc39.es/ecma262/2024/#sec-toboolean

Just Accepted

```
/*1*/ x = Math.random() <
    0.5;
/*2*/ y = !x;
/*3*/ if (x)  /*4*/ z = null
    ;
     else    /*5*/ z = 42;
/*6*/ z = !z; /*7*/
```

(a) An example JavaScript code.

$\mathbb{L}$	$\widehat{\Pi}$	$\widehat{\Sigma}$
1	$\bot$	
2	$\bot$	$\mathsf{x} \mapsto N$

$\mathbb{L}$	$\widehat{\Pi}$	$\widehat{\Sigma}$
3	$\bot$	$\mathsf{x} \mapsto N$ $\mathsf{y} \mapsto F$
4	$(3, \#\mathsf{t})$	$\mathsf{x} \mapsto \mathsf{t}$ $\mathsf{y} \mapsto F$
5	$(3, \#\mathsf{f})$	$\mathsf{x} \mapsto \mathsf{f}$ $\mathsf{y} \mapsto F$

$\mathbb{L}$	$\widehat{\Pi}$	$\widehat{\Sigma}$
6	$(3, \#\mathsf{t})$	$\mathsf{x} \mapsto \mathsf{t}$ $\mathsf{y} \mapsto F$ $\mathsf{z} \mapsto \mathsf{f}$
	$(3, \#\mathsf{f})$	$\mathsf{x} \mapsto \mathsf{f}$ $\mathsf{y} \mapsto F$ $\mathsf{z} \mapsto \mathsf{t}$

$\mathbb{L}$	$\widehat{\Pi}$	$\widehat{\Sigma}$
7	$(3, \#\mathsf{t})$	$\mathsf{x} \mapsto \mathsf{t}$ $\mathsf{y} \mapsto F$ $\mathsf{z} \mapsto \mathsf{t}$
	$(3, \#\mathsf{f})$	$\mathsf{x} \mapsto \mathsf{f}$ $\mathsf{y} \mapsto F$ $\mathsf{z} \mapsto \mathsf{f}$

(b) The path-sensitive truthiness analysis result.

Fig. 9. A simple example and its path-sensitive truthiness analysis result.

4.2.2 Abstract Semantics. The abstract semantics defines *how* this per-path information is propagated through each instruction of the CFG. The initial path-sensitive result $\hat{\xi}_\iota$ consists of a single mapping from the pair of the entry label ℓ_{entry} and $\bot$ to the initial abstract state $\hat{\sigma}_\iota$ initialized with all variables to N because they are not flipped.

$$\hat{\xi}_\iota = \{(\ell_{\mathsf{entry}}, \bot) \mapsto \hat{\sigma}_\iota\} \qquad \hat{\sigma}_\iota = \{x \mapsto N \mid x \in \mathbb{X}\}$$

The abstract semantics $\widehat{[\![\mathcal{G}]\!]} = \mathrm{lfp}\,\hat{F}$ of a given CFG $\mathcal{G}$ is the least fixed point of the abstract transfer function $\hat{F}$:

$$\hat{F}(\hat{\xi}) = \hat{\xi}_\iota \sqcup \left(\bigsqcup_{\ell \in \mathbb{L}} \bigsqcup_{\hat{\pi} \in \widehat{\Pi}} \widehat{[\![\mathsf{inst}(\ell)]\!]}_i(\ell, \hat{\pi}, \hat{\xi}(\ell, \hat{\pi})) \right)$$

where $\mathsf{inst}(\ell)$ is the instruction labeled by ℓ in the CFG, and $\widehat{[\![-]\!]}_i$ is the abstract semantics for instructions defined in Figure 8. The helper function $\mathsf{refine} : (\mathbb{L} \times \widehat{\Pi} \times \widehat{\Sigma}) \to (\mathbb{L} \times \mathbb{E} \times \mathbb{B}) \to \widehat{\Xi}$ refines abstract paths and states at conditional instructions. Its first triple $(\ell, \hat{\pi}, \hat{\sigma})$ is the current program point: the label of the conditional instruction, the abstract path under which it is reached, and the abstract state there. Its second triple (ℓ', e, b) is the branch being taken: the target label, the branch condition, and which of the two branches it is. We write $\widehat{[\![e]\!]}_e(\hat{\sigma}) = (\hat{v}, \hat{\sigma}')$ for the abstract value and the updated state that evaluating the condition produces. Intuitively, refine 1) produces no result entry when the point is unreachable ($\hat{v} = \bot$) or the taken branch is infeasible (the first two cases); 2) preserves the current path and state when the truthiness of the condition already agrees with the taken branch (the third case); and 3) otherwise starts a fresh abstract path (ℓ, b) (the last three cases), whose state additionally records the refined truthiness of the condition when the condition is a variable x (the fourth and fifth cases):

$$\mathsf{refine}(\ell, \hat{\pi}, \hat{\sigma})(\ell', e, b) = \begin{cases} \{\} & \text{if } (\hat{v} = \bot) \\ \{\} & \text{if } (\hat{v} = \mathsf{t} \wedge \neg b) \vee (\hat{v} = \mathsf{f} \wedge b) \\ \{(\ell', \hat{\pi}) \mapsto \hat{\sigma}'\} & \text{if } (\hat{v} = \mathsf{t} \wedge b) \vee (\hat{v} = \mathsf{f} \wedge \neg b) \end{cases} \begin{cases} \{(\ell', (\ell, b)) \mapsto \hat{\sigma}'[x \mapsto \mathsf{t}]\} & \text{if } \hat{v} \in \{F, N, \top\} \wedge e = x \wedge b \\ \{(\ell', (\ell, b)) \mapsto \hat{\sigma}'[x \mapsto \mathsf{f}]\} & \text{if } \hat{v} \in \{F, N, \top\} \wedge e = x \wedge \neg b \\ \{(\ell', (\ell, b)) \mapsto \hat{\sigma}'\} & \text{if } \hat{v} \in \{F, N, \top\} \wedge e \notin \mathbb{X} \end{cases}$$

The remaining rules of Figure 8 propagate this information through the other instructions and expressions. Assignments propagate the abstract value of their right-hand side. A constant or an object literal produces t or f according to the language semantics, a negation operator flips the truthiness, and other expressions produce N by default. Property operations and expressions with a potential side effect (e.g., function calls) initialize the abstract state with N for all variables, since a getter, a setter, or a callee may run arbitrary code and invalidate what the analysis has recorded.

Just Accepted

Example. To illustrate how the path-sensitive truthiness analysis works, we provide a simple example in Figure 9 and explain the analysis steps. The figure lists only the variables that have been assigned; the others remain N.

- **(Label 1)** The analysis begins at the entry label with the initial abstract state $\hat{\sigma}_\iota$, which maps every variable to N, under the base path $\bot$.
- **(Label 2)** Evaluating `x = Math.random() < 0.5` yields a boolean value whose truthiness is statically unknown; thus, `x` is mapped to N (not flipped).
- **(Label 3)** The assignment `y = !x` flips the truthiness of `x`, so `y` is mapped to F (flipped).
- **(Label 4)** When `x` appears in the condition of the `if` statement, the analysis refines paths. In the true branch, the path is $(3, \texttt{\#t})$ with $\texttt{x} = \texttt{t}$.
- **(Label 5)** On the other hand, in the false branch, the path is $(3, \texttt{\#f})$ with $\texttt{x} = \texttt{f}$.
- **(Label 6)** Although outside the `if` statement, the analysis preserves the refined abstract paths. Here, `z` is mapped to f under $(3, \texttt{\#t})$ and to t under $(3, \texttt{\#f})$, respectively, because `null` is a falsy value but `42` is a truthy value.
- **(Label 7)** The assignment `z = !z` flips the abstract values of `z` in each path.

4.3 Analysis-Based Flow-Refinement

Using the analysis result $\hat{\xi} = \widehat{[\![\mathcal{G}]\!]}$ of the CFG $\mathcal{G}$ being refined, we refine CFGs in three steps to construct consistent POGs. Figure 10 shows the refinement of the CFG of the bundled code in Figure 1b step by step, where a dotted edge denotes a removed edge and thick red lines denote the modified or added part.

Step 1: Branch Flipping. Transpilers often flip branches by negating the condition or its incoming expressions, as in:

```
// 1 non-flipped (N) and 2 flipped (F) values
if (y) { x = e1 == e2; } else if (z) { x = !e3; } else { x = !e4; }
if (x) { thenStmt; } else { elseStmt; }
```

The truthiness of e_1 == e_2 flows to the condition x without flipping, while the truthiness of the two expressions e_3 and e_4 is flipped before it flows to x. SWC flips the branches (`thenStmt` and `elseStmt`) of the `if`-statement by negating all expressions that flow into the condition x.

```
// 2 non-flipped (N) and 1 flipped (F) values
if (y) { x = e1 != e2; } else if (z) { x = e3; } else { x = e4; }
if (x) { elseStmt; } else { thenStmt; }
```

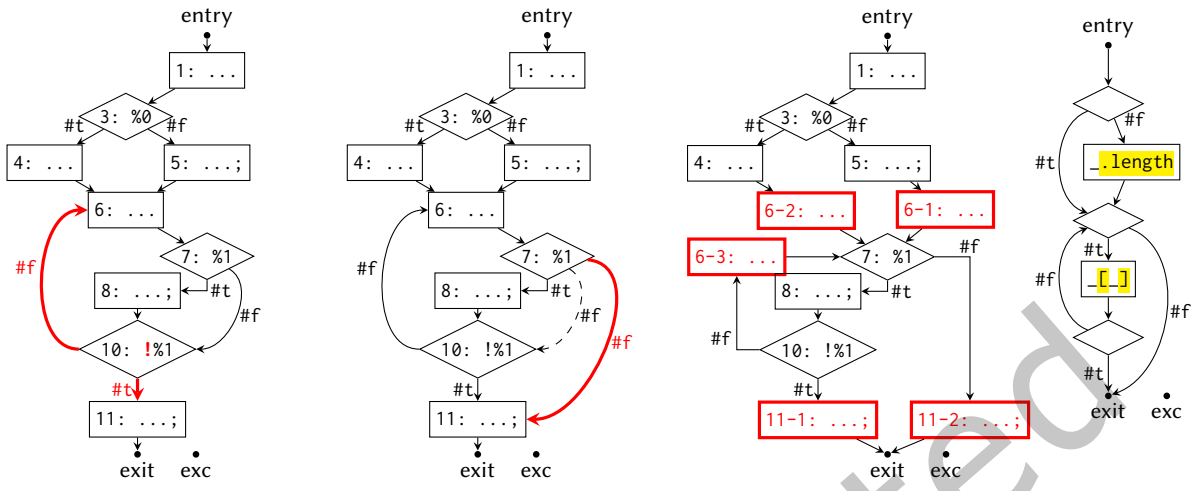
For consistent branch flipping status, we flip a branch if *more flipped* values flow into it than non-flipped ones. Flipping negates the condition e of the branch and swaps its two branch labels, which together preserve the semantics of the conditional instruction. For example, one flipped value flows into the condition `%1` at `10` from the true branch of `7`, and no non-flipped value does:

$$(\texttt{\%1} \mapsto F) \in \hat{\xi}(\texttt{10}, (\texttt{7}, \texttt{\#t}))$$

Thus, our approach flips the branch at `10` by negating the condition `%1` to `!%1` as shown in Figure 10a, where the true branch now leads to `11` and the false branch back to `6`.

Step 2: Branch Bypassing. This step undoes minification patterns that merge nested branches into short-circuit logical expressions, which leave behind branches that are always taken in one direction on a given path. If a

Just Accepted



(a) Result of Branch Flipping. (b) Result of Branch Bypassing. (c) Result of Path Cloning. (d) POG of (c).

Fig. 10. Step-by-step analysis-based flow refinement of the CFG of bundled code in Figure 2a and its POG.

condition is always truthy (or falsy) on a path, we bypass the branch by linking its in-edge directly to the true (or false) out-edge. For instance, along the path from the branch at 7 (false) to the branch at 10, the analysis shows that %1 is falsy:

$$(\%1 \mapsto \mathsf{f}) \in \hat{\xi}(10, (7, \#\mathtt{f}))$$

The condition !%1 of 10, negated by branch flipping, is therefore always truthy on this path. Thus, we can bypass the branch at 10 by connecting the false branch of 7 to 11 directly as shown in Figure 10b. This step covers a transformation that minifiers commonly apply using logical expressions: `if(x){ if(y) return z; }` → `if (x && y) return z;`

Step 3: Path Cloning. This step undoes minification patterns that merge duplicated branch bodies into a single shared block, which makes property operations appear once instead of once per branch. If normal instructions are reachable from multiple abstract paths, we clone them for each abstract path. For example, the instructions at 6 are reachable from three different abstract paths: the false and true branches of 3 and the false branch of 10 according to the analysis result:

$$\hat{\xi}(6, (3, \#\mathtt{f})) \neq \bot \qquad \hat{\xi}(6, (3, \#\mathtt{t})) \neq \bot \qquad \hat{\xi}(6, (10, \#\mathtt{f})) \neq \bot$$

Similarly, the instruction at 11 is reachable from two paths:

$$\hat{\xi}(11, (7, \#\mathtt{f})) \neq \bot \qquad \hat{\xi}(11, (10, \#\mathtt{t})) \neq \bot$$

Thus, we clone the instructions at 6 and 11 for each abstract path as shown in Figure 10c. While path cloning does not affect the POG in this example, it is helpful to construct consistent POGs for the following minification pattern: `if(y) x.p = e1; else x.p = e2;` → `x.p = y ? e1 : e2;` After cloning, both versions have a property write of p in each branch, making their POGs consistent.

After the three refinement steps, the CFGs of the original and bundled code in Figure 1 converge to the same normal form, and the POGs extracted from them become identical, resolving the inconsistency illustrated in §2.2.

Finally, Debun hashes the POG of each function into a single *fingerprint*, which the detection in §6 compares. Starting from the exit node, we hash each node from its node type, its property operations, and the hashes of its

Just Accepted

successors, so that the entry node's hash summarizes the entire POG. A back edge from a loop contributes a fixed marker instead of the hash of its target, keeping the computation well founded. Visiting the true branch of a conditional node before the false one makes the hash deterministic, so identical POGs always receive the same fingerprint. Memoization with unique node IDs avoids recomputing shared subgraphs. Every hash is computed with rapidhash,[4] a fast non-cryptographic hash function.

5 General Function Inlining for Preprocessing

Minifiers, which are typically integrated into bundlers, apply diverse transformations to reduce file sizes. While POG fingerprints are highly resilient to most transformations (e.g., identifier mangling and dead code elimination), their robustness is compromised by transformations that cross function boundaries. A prevalent example of such transformations is *function inlining*, which injects the body of a callee function into its call site when the function is used only once. For instance, consider the following excerpt from Lodash.js v4.17.21:

```
function isLength(value) {
  return typeof value == "number" && value > -1 && value % 1 == 0 && value <= MAX_SAFE_INTEGER;
}
function isArrayLike(value) {
  return value != null && isLength(value.length) && !isFunction(value);
}
```

When processed by SWC, `isLength` is inlined at its unique call site `isLength(value.length)` within `isArrayLike`:

```
function isArrayLike(r){var i;return null!=r&&
"number"==typeof(i=r.length)&&i>-1&&i%1==0&&i<=MAX_SAFE_INTEGER&&!isFunction(r)}
```

Consequently, while the original code yields two distinct POG fingerprints (one for each function), the transpiled code yields only a single, merged fingerprint. This merger prevents the fingerprints of the original and the transpiled code from matching. Which functions are actually inlined is moreover far from consistent: the precision and the scope of the analysis that identifies single-use functions differ across minifiers, across their versions, and across their option settings. Inlining also interacts with the other transformations applied around it, so the same function may be inlined in one build and left alone in another.

To make POG fingerprints robust against such inlining, this section introduces a *function inlining* preprocessing pass prior to POG generation. Specifically, this pass operates in two steps: (1) it identifies functions invoked from *exactly one* call site without having other possible references, and (2) it eliminates the identified single-use functions by inlining their bodies at the unique call sites, thereby canonicalizing the code structure. Note that this inlining is performed only within a single source file, not across modules.

One might instead run a bundler over the library sources and fingerprint its output. That output matches a website only if we choose the same bundler, version, and options that the website used. A deployed artifact reveals none of the three, and enumerating them is the exhaustive-transpilation approach that §2.1 shows to be unscalable. Running a bundler on the website side does not help either, since it cannot undo the transformations the site has already applied. Our pass instead inlines every single-use function on both sides, so the two agree whether or not the minifier inlined a given function. It applies the transformation as widely as it can rather than reproducing any minifier's decision, and therefore inlines functions that no minifier would have touched. This is harmless because both sides are treated alike. Where the pass cannot establish its premises, the function

[4] https://github.com/Nicoshev/rapidhash

Just Accepted

```
function f1(...) { ... }
f2(42); // unique use of f1
f3 = function (...) { ...
    };
let x1 = f4;
x2("a"); // unique use of
    f3
```

```
function g1() { ... }
function h1() { ... }
if (...) g2 = h2;
g3(); // no inlining (multi
    defs)
h3(); // no inlining (multi
    uses)
```

$$\text{Def} = \begin{cases} f_2 \mapsto \{f_1\} \\ f_4 \mapsto \{f_3\} \\ x_2 \mapsto \{x_1\} \\ g_3 \mapsto \{g_1, g_2\} \\ h_2 \mapsto \{h_1\} \\ h_3 \mapsto \{h_1\} \end{cases} \quad \text{Use} = \begin{cases} f_1 \mapsto \{f_2\} \\ f_3 \mapsto \{f_4\} \\ x_1 \mapsto \{x_2\} \\ g_1 \mapsto \{g_3\} \\ g_2 \mapsto \{g_3\} \\ h_1 \mapsto \{h_2, h_3\} \end{cases}$$

(a) JavaScript program

(b) Def-use dataflow analysis results.

Fig. 11. Example JavaScript program and its def-use dataflow analysis results.

fails to match rather than matching a wrong library (§8). The same reasoning motivates the analysis-based flow refinement in §4.3 rather than replaying the branch rewritings of a specific bundler.

5.1 Exactly-Once Analysis

To find the functions that are used *exactly once*, we first assign unique labels to each variable occurrence in the JavaScript AST, enabling precise tracking of variable definitions and uses. A labeled variable $x_\ell \in \mathbb{X}_\ell$ denotes the occurrence of a variable name x at a specific program point identified by the label ℓ. We partition the occurrences in $\mathbb{X}_\ell$ into two disjoint domains: the set $\mathbb{X}_\ell^{\text{def}}$ of *definition occurrences*, containing the occurrences on the left-hand side of variable-defining constructs (e.g., variable/function/class declarations, parameters, and assignment expressions), and the set $\mathbb{X}_\ell^{\text{use}} = \mathbb{X}_\ell \setminus \mathbb{X}_\ell^{\text{def}}$ of *use occurrences*. For instance, the assignment expression $x = x + x$ is represented in the labeled form as $x_1 = x_2 + x_3$, where $x_1 \in \mathbb{X}_\ell^{\text{def}}$ is a definition occurrence, while $x_2, x_3 \in \mathbb{X}_\ell^{\text{use}}$ are use occurrences. An update or compound assignment such as `x++` or `x += 1` both reads and writes x at the same program point, so we label its read and its write as separate occurrences.

Next, we perform a standard *def-use data-flow analysis*. Following the classical iterative algorithm [2], this analysis computes the least fixpoint over the transfer functions defined by the gen and kill sets of variable definitions and uses. To ensure accurate tracking of variable bindings, our analysis specifically accounts for JavaScript's complex scoping rules, including recursive functions, function/variable hoisting (`var`), and lexical block scoping (`let` and `const`). Occurrences of the same name in different scopes are therefore never conflated. The analysis results in a set of def-use pairs, which are used to construct the following two mappings between the two occurrence domains:

$$\text{Def} : \mathbb{X}_\ell^{\text{use}} \to \mathcal{P}(\mathbb{X}_\ell^{\text{def}}) \qquad \text{Use} : \mathbb{X}_\ell^{\text{def}} \to \mathcal{P}(\mathbb{X}_\ell^{\text{use}})$$

where Def maps each use occurrence to the set of definition occurrences that potentially reach it, and Use maps each definition occurrence to the set of use occurrences it may flow into. For example, Figure 11 illustrates a sample JavaScript program alongside its corresponding def-use analysis results. In this example, the variable reference f_2 is uniquely defined by the declaration at f_1 (i.e., $\text{Def}(f_2) = \{f_1\}$). In contrast, the reference g_3 may be defined by either g_1 or g_2 depending on the execution path due to the conditional assignment, resulting in a non-singleton set: $\text{Def}(g_3) = \{g_1, g_2\}$. Note that the unique def-use flows we derive below correspond to def-use chains between a single static definition and a single use, as in static single-assignment (SSA) form; we nevertheless perform the analysis directly on the labeled AST rather than converting to SSA, since JavaScript's function/variable hoisting and lexical scoping make AST-level def-use tracking more direct without introducing ϕ-nodes.

Just Accepted

Unique Def-Use Flow. Using the computed def-use mappings, we formalize the *unique def-use flow* relation with the following two inference rules for direct and indirect flows:

$$\textsc{U-Direct}\ \frac{\neg\mathsf{isExposed}(\mathsf{bind}(x_i)) \qquad \mathsf{Def}(x_j) = \{x_i\} \qquad \mathsf{Use}(x_i) = \{x_j\}}{x_i \xrightarrow[u]{\epsilon} x_j} \qquad \textsc{U-Indirect}\ \frac{x_i \xrightarrow[u]{\epsilon} x_j \qquad (y_k := x_j) \in P \qquad y_k \xrightarrow[u]{\overline{y'_l}} y''_m}{x_i \xrightarrow[u]{y_k :: \overline{y'_l}} y''_m}$$

Here, $\overline{A}$ is a sequence of A, ϵ is the empty sequence, $A :: \overline{A'}$ is the sequence obtained by prepending an element A to a sequence $\overline{A'}$, and P is the analyzed program (i.e., the labeled AST). The judgment $x_i \xrightarrow[u]{\overline{y_j}} z_k$ states that the definition occurrence $x_i \in \mathbb{X}_\ell^{\mathsf{def}}$ *uniquely* reaches the use occurrence $z_k \in \mathbb{X}_\ell^{\mathsf{use}}$ through the intermediate definition occurrences $\overline{y_j} \in (\mathbb{X}_\ell^{\mathsf{def}})^*$ of alias assignments. Each step along the chain is a one-to-one def-use pair, so the value defined at x_i can be used only at z_k. The base case (U-Direct) is the direct flow with an empty chain, where x_i is the sole definition reaching x_j, x_j is the only use of x_i, and the binding of x_i is not exposed. The predicate isExposed is defined on *bindings* rather than on individual occurrences, since it is a property of the variable and not of one of its occurrences. The def-use analysis resolves each occurrence to its binding, written $\mathsf{bind}(x_\ell)$, and a binding is *global* when it is declared in the global scope of script code, *exported* when it is exported from module code, and *local* otherwise; only the first two are exposed. An exposed binding may be referenced by external scripts or modules, so we conservatively assume that it has uses we cannot observe and never inline it. The inductive case (U-Indirect) captures indirect flow through aliasing. The notation $(y_k := e) \in P$ indicates that the definition occurrence $y_k \in \mathbb{X}_\ell^{\mathsf{def}}$ is defined with an explicit syntactic initializer e in P. For instance, a function declaration `function f() {}` is represented as $\mathtt{f} := \mathtt{function}\ ()\ \{\}$, whereas parameters are excluded, since they are initialized by runtime arguments. An *alias assignment* $(y_k := x_j)$ is the special case of this form whose initializer is a lone use occurrence $x_j \in \mathbb{X}_\ell^{\mathsf{use}}$. In the third premise $y_k \xrightarrow[u]{\overline{y'_l}} y''_m$, the sequence $\overline{y'_l}$ may itself be empty (ϵ), in which case the premise is derived by U-Direct. Thus, the recursion bottoms out at a direct flow, and each application of U-Indirect extends the alias chain by one element.

In Figure 11, assuming that all its functions are non-global, we identify two unique def-use flows, one direct and one through the assignment at $\mathtt{x}_1$:

$$\mathtt{f}_1 \xrightarrow[u]{\epsilon} \mathtt{f}_2 \qquad \text{and} \qquad \mathtt{f}_3 \xrightarrow[u]{[\mathtt{x}_1]} \mathtt{x}_2$$

In contrast, $\mathtt{g}_3$ and $\mathtt{h}_3$ establish no unique def-use flow: two definitions reach $\mathtt{g}_3$ ($\mathsf{Def}(\mathtt{g}_3) = \{\mathtt{g}_1, \mathtt{g}_2\}$), and although $\mathtt{h}_1$ is the only definition reaching $\mathtt{h}_3$, it also reaches $\mathtt{h}_2$ ($\mathsf{Use}(\mathtt{h}_1) = \{\mathtt{h}_2, \mathtt{h}_3\}$).

5.2 Function Inlining Transformation

Using the unique def-use flows of §5.1, we now define when a function is inlinable at a call site and how we inline it.

Inlinability Criterion. We define that a function declaration x_i is *inlinable* at a call site z_k via $\overline{y_j}$ (i.e., $\mathsf{inlinable}(x_i, \overline{y_j}, z_k)$) if there is a unique def-use flow from x_i to z_k through a sequence of intermediate variable assignments $\overline{y_j}$. A *call site* $z_k(\ldots) \in P$ is likewise a distinct syntactic form: a call expression whose callee is a use occurrence $z_k \in \mathbb{X}_\ell^{\mathsf{use}}$. To ensure termination and avoid recursive inlining, the call site z_k must not reside within the body of the function

Just Accepted

declared at x_i.

$$\mathsf{inlinable}(x_i, \overline{y_j}, z_k) \iff \overbrace{(x_i := \texttt{function}(\ldots)\,\{\,B\,\}) \in P}^{\text{function declaration}} \;\wedge\; \overbrace{x_i \xrightarrow[u]{\overline{y_j}} z_k}^{\text{unique def-use flow}} \;\wedge\; \overbrace{z_k(\ldots) \in P}^{\text{call site}} \;\wedge\; \overbrace{z_k \notin B}^{\text{non-recursive call}}$$

Both flows of Figure 11 satisfy the remaining conditions, so $\mathsf{inlinable}(\mathtt{f}_1, \epsilon, \mathtt{f}_2)$ and $\mathsf{inlinable}(\mathtt{f}_3, [\mathtt{x}_1], \mathtt{x}_2)$ hold.

Transformation. Once the criterion is satisfied for a function declaration $x_i := \texttt{function}(\ldots)\,\{\,B\,\}$, we perform a source-to-source transformation to collapse the function boundary. We first remove all intermediate variable assignments $\overline{y_j}$ along the unique def-use flow from x_i to z_k, and then alpha-rename the body to prevent unintended variable capture. Minifiers (e.g., SWC) typically inline a function by substituting its parameters with the argument expressions at the call site, provided specific parameter usage patterns are met. To handle all inlinable functions uniformly instead, we wrap the original function as an Immediately Invoked Function Expression (IIFE) and invoke it with the arguments from the call site z_k. The declaration x_i is then removed, since its only use has been replaced. Applied to the program of Figure 11, the pass removes the declaration at $\mathtt{f}_1$, the assignment at $\mathtt{f}_3$, and the intermediate assignment at $\mathtt{x}_1$:

```
(function (...) { ... })(42);    // f1 inlined at f2
(function (...) { ... })("a");   // f3 inlined at x2
```

The functions g and h are left unchanged, since neither has a unique def-use flow.

Transparent IIFEs. The IIFE remains in the rewritten AST, but the subsequent POG Generation phase (§4) does not treat it as a function boundary. Instead, Debun expands it inline during CFG construction (§4.1.1): the parameters become assignment instructions that bind the argument expressions in their original order, and the body is spliced into the caller's CFG at the call site. Each `return` statement in the body becomes an assignment to the temporary variable holding the call's result, with its next label pointing to the instruction following the IIFE. All other function calls remain ordinary call expressions with potential side effects. This expansion lets the property operations of the inlined body appear in the caller's POG in their original order, as if the code were linearized. It also preserves the original call's evaluation order and side effects, since the parameter assignments precede the body.

Single-Pass Application and Termination. Since the transformation rewrites the program, one could re-run the analysis on the result and inline again. This is unnecessary, because inlining cannot create a new inlinable site. It removes a declaration and the alias assignments along one flow, and replaces one call with an IIFE; none of these gives another function a unique def-use flow it did not already have. Splicing a body into a call site can only remove candidates, by bringing a call inside the body of the function it calls. We therefore apply the transformation in a *single pass*: the def-use analysis (§5.1) identifies all inlinable call sites at once and inlines each exactly once, so the pass terminates and a second pass finds nothing more. Mutually recursive functions need no iteration either. Our criterion allows partial inlining within a mutually recursive group, provided the call site is external to the function's own body:

```
function f1(x) { return g1(x); }     // not inlinable - two uses f2, f3
function g2(x) { return h1(x * 2); } // inlinable at g1
function h2(x) { return f2(x + 1); } // inlinable at h1
f3(...);
```

Just Accepted

Here, g_2 and h_2 are inlined at g_1 and h_1 within the same pass, which collapses the group into a direct recursion on `f`:

```
function f1(x) { return (function (...) { ... f2(...) ... })(x); }
f3(...);
```

The only remaining call to `f` is f_2, which now sits inside f_1's own body and is therefore excluded by the non-recursive-call condition. The collapse always stops at such a member. A group entered from outside, as `f` is here through f_3, has a member with a second use that is not inlinable, and a group that nothing outside calls is unreachable.

6 Library and Version Detection

Debun detects libraries and their specific versions on a target website using a *similarity score* function (score). This function computes the similarity between the set of fingerprints collected from the website and that of each library version stored in the database. Here, a *fingerprint* is the hash of a single function's POG (§4), and the database maps each *library-version pair* (L, v) of a library L and one of its versions v to the set of fingerprints of all functions in that version. Detection is performed by *hash equality* rather than by structurally comparing POGs. This section introduces two similarity scoring functions: 1) the *uniform score* (§6.1) and 2) the *module-dependency-aware score* (§6.2), each used with its own threshold parameter: ψ_u for the former and ψ_m for the latter. The two parameters bound different quantities and are therefore tuned separately, as we explain where each score is defined. The uniform score is the natural baseline that treats a library as a single unit, and its failure on partially imported libraries motivates the module-dependency-aware score. Debun therefore uses the latter, and §7.4.2 quantifies the gap between the two.

6.1 Uniform Score

The *uniform scoring* function score_u computes the similarity score for each library-version pair (L, v) by calculating the ratio of fingerprints matched on the target website to the total number of fingerprints in (L, v):

$$\text{score}_u(L, v) = \frac{(\#\text{ Matched Fingerprints of } (L, v))}{(\#\text{ Total Fingerprints of } (L, v))}$$

Given a *threshold parameter* $\psi_u \in [0, 1]$, we conclude that a library L is present on the target website if at least one of its versions achieves a score greater than or equal to ψ_u:

$$L \text{ is detected} \quad \Longleftrightarrow \quad \exists v.\ \text{score}_u(L, v) \geq \psi_u$$

The threshold ψ_u bounds how large a fraction of a version's fingerprints may be absent from the website, whether because the build removed the corresponding functions or because a transformation altered them beyond what our normalization recovers. We report its tuned value in §7.4.2. For example, if a version (L, v) has 20 fingerprints in the database and 17 of them are matched on the target website, then $\text{score}_u(L, v) = 17/20 = 85\%$, and L is reported as present whenever $\psi_u \leq 85\%$. For each detected library, we determine its version via *unique fingerprints* and similarity scores. A fingerprint is considered *version-unique* if it is exclusively found in a single version of a library, and *library-unique* if it is not shared with any other library. The version selection process prioritizes versions by applying the following criteria in order:

(1) Select the version(s) containing *version-unique* fingerprints.
(2) Select the version(s) with the highest number of *library-unique* fingerprints.
(3) Select the version(s) with the highest overall similarity score.

Just Accepted

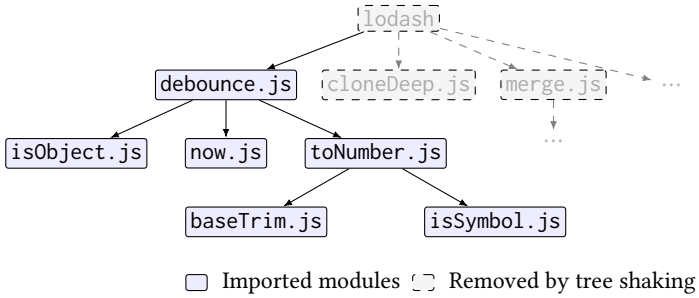


Fig. 12. Tree shaking keeps only the debounce module of Lodash.js and the modules it depends on.

Filtering Overlapping Functions. Beyond the bundler-injected boilerplate already removed when the database is built (§3.2), the uniform score suffers from overlapping functions between libraries, which arise when libraries partially include others (i.e., *shrinkwrapped clones* [34, 36]) or share common code patterns. To mitigate them, we apply *code segmentation* [34] based on a *birth* date (the release date of a version). If more than 30% of functions overlap between two libraries, we retain only the copies from the library version with the earliest birth date. We set this overlap threshold conservatively at 30%, three times stricter than the 10% reuse ratio empirically selected by Centris [34], which observed that widely shared generic code generally accounts for less than 10% of a project. Only clear shrinkwrapped cloning, rather than incidentally shared utility code, therefore triggers the filtering.

Limitations of Uniform Score. JavaScript libraries consist of multiple modules, and developers often import only a required subset. Bundlers then perform *tree shaking* [19] to eliminate unused modules. However, the uniform score treats a library as a single unit, failing to account for fingerprints removed by tree shaking. For example, if a website imports only the `debounce` module from the Lodash.js library:

```
import debounce from "lodash/debounce";
```

During the build process, the bundler includes only the `debounce` module and its dependencies in the final bundle, while the rest of the Lodash.js library is removed by tree shaking. Thus, the uniform score of Lodash.js on the website would be extremely low, since most of the library's modules (e.g., `merge`, `cloneDeep`) are not in the bundle (Figure 12). Lowering ψ_u far enough to catch such cases does not help, because the same low bar then accepts weak evidence for every library and costs precision (§7.4.2). The problem is that the denominator counts modules the website never imported, so the fix must change what the score is normalized by.

6.2 Module-Dependency-Aware Score

To address the limitations of the uniform score, we introduce a *module-dependency-aware scoring* function (score_m) that accounts for the module-level structure of libraries and the effects of tree shaking. The uniform score attempts to answer the question, "*Is library X present on this site?*" by evaluating the entire package as a single monolithic unit. In contrast, the module-dependency-aware score asks, "*Which modules of library X are imported on this site?*" and computes the similarity score based on only the imported modules instead of the whole library. Every threshold check below uses the single parameter ψ_m, which bounds how many fingerprints are expected to survive one bundling step.

6.2.1 Module-Dependency Graph (MDG). Libraries delivered via Content Delivery Networks (CDNs) are typically distributed as pre-bundled files without module structures, obscuring which specific modules are present on

Just Accepted

the target website. Conversely, libraries published as npm packages retain multiple files (modules) connected with explicit module dependency information. However, npm packages often contain non-core files (e.g., tests, examples) that are never imported into production bundles. To combine the strengths of both formats, we decompose the CDN bundle into modules leveraging npm packages as a grouping schema. We then construct a *module-dependency graph* (MDG) $G = (\mathcal{N}, \mathcal{E})$ for each library version based on the npm dependency structure and the presence of its modules in the CDN bundle. We use the following hash-set notation: $H_{\mathsf{npm}}(m)$ is the set of function fingerprints collected from a module m of the npm package, H_{cdn} is the set of function fingerprints collected from the pre-bundled CDN file of the same library version, and H_{web} is the set of function fingerprints collected from the target website.

- A **node** $m \in \mathcal{N}$ represents a module (file) from the npm package, included only if its function-level POG hashes $H_{\mathsf{npm}}(m)$ have a sufficient intersection with the CDN bundle's hash set H_{cdn}:

$$m \in \mathcal{N} \iff \frac{|\, H_{\mathsf{npm}}(m) \cap H_{\mathsf{cdn}} \,|}{|H_{\mathsf{npm}}(m)|} \geq \psi_m$$

  A module with an empty $H_{\mathsf{npm}}(m)$, for which this ratio is undefined, is excluded from $\mathcal{N}$. This check crosses the library's own build, from its npm sources to its CDN bundle, whereas the two checks below cross the website's build. Each node m stores its hash set $H(m) = H_{\mathsf{npm}}(m) \cap H_{\mathsf{cdn}}$: the set of function hashes from the npm module that are present in the CDN bundle.
- An **edge** $(m_i, m_j) \in \mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ represents a statically resolvable dependency (e.g., `import` or `require` statements) directed from an importing module m_i to an imported module m_j. This edge is added only if both m_i and m_j exist in $\mathcal{N}$ according to the threshold criterion above. We exclude dynamic imports (e.g., `import()` with non-literal arguments) because their dependencies cannot be statically resolved.

Modules with an empty $H_{\mathsf{npm}}(m)$ are typically *re-export-only*, containing only `import` and `export` statements. We *bypass* them to preserve transitive dependencies (e.g., removing B from $A \to B \to C$ induces $A \to C$).

Module-Level Clone Filtering. To resolve *shrinkwrapped clones* [34, 36] based on release dates, the uniform score discards the functions that a library shares with an older one whenever their overlap exceeds 30% (§6.1 explains this threshold choice). Under the module-dependency-aware score, we apply this filtering at the module level. We summarize each module m using a single *composite hash* by sorting, concatenating, and rehashing its function-level hashes $H(m)$. When a module's composite hash matches that of a module in an earlier-published library, we treat the earlier one as the original and filter out the copy from the later library. If a module is removed, we likewise preserve transitive dependencies among the remaining modules by bypassing it. As a limitation, modified or partially copied modules yield different composite hashes and thus remain unfiltered. This can occasionally lead to missed clones and false positives.

Figure 13 shows an MDG, in which each circle is a module and each directed edge a dependency. Its remaining annotations, which mark candidate entries and label modules and closures as detected or undetected, are explained below. We write $G(L, v)$ for the MDG of library L version v.

6.2.2 MDG-based Library and Version Detection. Developers can import any module as an entry point, causing tree shaking to retain only its dependent modules in the final bundle. Given an MDG $G = (\mathcal{N}, \mathcal{E})$ and a target website's hash set H_{web}, we identify which entry-rooted module closures are present using the following steps. Algorithm 1 specifies the complete detection procedure as pseudocode, and we explain each step below. Four notational conventions keep it compact. We write $\mathit{ratio}(H) = |H \cap H_{\mathsf{web}}| \,/\, |H|$ for the match ratio of a hash set H against the website, we lift H to sets of modules as $H(S) = \bigcup_{m \in S} H(m)$, we lift reach to sets of entries as $\mathsf{reach}(S) = \bigcup_{M \in S} \mathsf{reach}(M)$, and we index a quantity by (L, v) when it is computed on the MDG $G(L, v)$ and later compared across versions.

Just Accepted

Algorithm 1: MDG-based library and version detection.

Input: website hash set H_{web}; MDGs $G(L, v)$ of all library versions; threshold ψ_m
Output: set R of detected libraries paired with their selected versions

1 $R \leftarrow \emptyset$;
2 **foreach** *library* L *in the database* **do**
3 **foreach** *version* v *of* L *with its MDG* $G(L, v) = (\mathcal{N}, \mathcal{E})$ **do**
4 $\mathcal{N}_\exists \leftarrow \{ m \in \mathcal{N} \mid \mathit{ratio}(H(m)) \geq \psi_m \}$; // module detection
5 partition $G(L, v)$ into strongly connected components $\mathcal{N}^{\text{SCC}}$;
6 $\mathcal{N}^{\text{Entry}} \leftarrow$ SCCs with a detected module but no detected ancestor; // candidates
7 $\mathcal{N}^{\text{Entry}}_\exists(L, v) \leftarrow \{ M \in \mathcal{N}^{\text{Entry}} \mid \mathit{ratio}(H(\text{reach}(M))) \geq \psi_m \}$; // entry confirmation
8 $\text{score}_m(L, v) \leftarrow \mathit{ratio}(H(\text{reach}(\mathcal{N}^{\text{Entry}}_\exists(L, v))))$, or 0 if $\mathcal{N}^{\text{Entry}}_\exists(L, v) = \emptyset$;
9 **if** $\exists v.\ \mathcal{N}^{\mathit{Entry}}_\exists(L, v) \neq \emptyset$ **then** // library detection
10 add (L, v^*) to R for the version(s) v^* maximizing $\text{score}_m(L, v^*)$; // version selection
11 **return** R

(1) **Module Detection** (line 4): We determine if each module $m \in \mathcal{N}$ is present on the website by its overlap with H_{web}, so the detected module set is:

$$\mathcal{N}_\exists = \{ m \in \mathcal{N} \mid \mathit{ratio}(H(m)) \geq \psi_m \}$$

(2) **Candidate Entry Selection** (lines 5–6): We identify *candidate entry* modules that could serve as roots of imported closures:
  (a) First, we partition $\mathcal{N}$ into strongly connected components (SCCs), denoted $\mathcal{N}^{\text{SCC}}$, to handle cyclic dependencies.
  (b) Then, we define the candidate entry set $\mathcal{N}^{\text{Entry}} \subseteq \mathcal{N}^{\text{SCC}}$ as the SCCs containing at least one detected module but having no detected ancestors in the MDG.

(3) **Closure Computation and Entry Detection** (line 7): The *closure* of an entry $M \in \mathcal{N}^{\text{Entry}}$, denoted as $\text{reach}(M)$, is the set of all modules reachable from M in the MDG. Because H lifts by *set* union, a fingerprint shared by multiple modules (e.g., a common utility function) is counted only once. We then confirm an entry M by evaluating its entire closure against H_{web}, which gives the confirmed entry set:

$$\mathcal{N}^{\text{Entry}}_\exists = \{ M \in \mathcal{N}^{\text{Entry}} \mid \mathit{ratio}(H(\text{reach}(M))) \geq \psi_m \}$$

Based on these detected entries and their closures, we finalize library and version detection.

- **Library Detection** (line 9): A library L is detected on the website if some version of it contains a confirmed entry:

$$L \text{ is detected} \quad \Longleftrightarrow \quad \exists v.\ \mathcal{N}^{\text{Entry}}_\exists(L, v) \neq \emptyset$$

- **Version Detection** (line 10): The module-dependency-aware score of a version is the match ratio of the closures of all its confirmed entries (line 8):

$$\text{score}_m(L, v) = \mathit{ratio}(H(\text{reach}(\mathcal{N}^{\text{Entry}}_\exists(L, v))))$$

Just Accepted

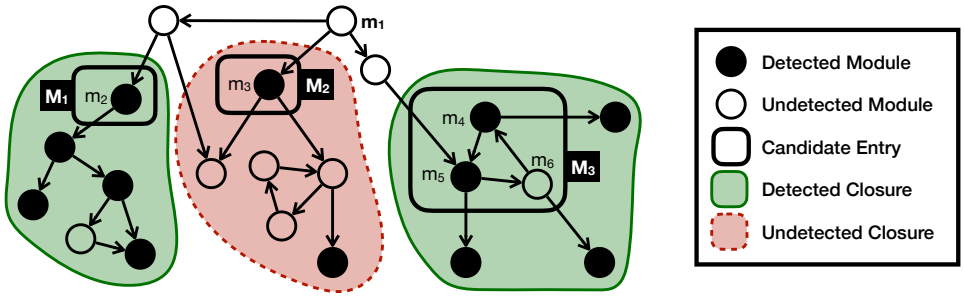


Fig. 13. An example of a module-dependency graph (MDG) and the detection process with a website hash set H_{web}.

For each detected library L, we select the version(s) maximizing this score, and if multiple versions tie for the maximum, we select all of them.

$$\text{version of } L \quad = \quad \arg\max_{v} \ \text{score}_m(L, v)$$

Unlike the uniform score (§6.1), this selection does not consider version-unique or library-unique fingerprints. The confirmed closures are the only part of a version for which the website provides evidence, and a fingerprint unique to a version but outside those closures was removed by tree shaking, so its absence says nothing about the version. As a limitation, the match ratio is normalized by closure size, so a version whose confirmed closures are small can tie with or outscore one that matches more functions overall, which widens the candidate ranges reported in §7.6.

Example. Figure 13 illustrates the detection process on an MDG using a website hash set H_{web}. Although m_1 is the library's top-level module, developers can import any module as an entry point. The algorithm first identifies 12 present modules (filled nodes) and 9 missing ones (empty nodes). Three SCCs ($M_1 = \{m_2\}$, $M_2 = \{m_3\}$, and $M_3 = \{m_4, m_5, m_6\}$) are selected as candidate entries because they contain detected modules but have no detected ancestors. Next, the closures of M_1 and M_3 are confirmed (green areas, solid borders) due to sufficient intersection with H_{web}, whereas M_2's closure fails this threshold (red area, dashed border). For instance, with $\psi_m = 84\%$, suppose the modules in reach(M_2) contain 12 fingerprints in total but only 4 of them appear in H_{web}; the match ratio of M_2's closure is then $4/12 \approx 33\% < \psi_m$, so M_2 is rejected even though its entry module m_3 itself is detected. With M_1 and M_3 confirmed as entries, the library is successfully detected. Finally, the version is determined by the match ratio of these confirmed closures. For example, suppose reach(M_1) and reach(M_3) together contain 14 function fingerprints, of which 12 appear in H_{web}; the match ratio for this version is then $12/14 \approx 86\%$, which clears ψ_m. Debun computes this ratio for every candidate version and selects the version(s) with the highest ratio (all of them, in case of a tie).

7 Evaluation

We evaluate Debun to answer the following research questions:

- RQ1. **Performance:** How much time and memory does Debun need to detect libraries, and how does its cost compare with that of existing tools?
- RQ2. **Ablation Study:** How do different POG generation techniques impact fingerprint consistency and accuracy, and how do different library scoring functions affect the overall detection performance?
- RQ3. **Library Detection:** Does Debun outperform existing tools such as LDC, PTdetector, and D-Bundlr on library detection in real-world websites?

Just Accepted

- RQ4. **Version Detection:** Does Debun outperform existing tools such as LDC and Aletheia on library version detection in real-world websites, and PTV on a controlled single-library corpus?

We first report the detection cost of Debun (RQ1), then justify our configuration choices with an ablation study (RQ2), and finally compare Debun against state-of-the-art tools under that configuration (RQ3 and RQ4). All experiments were conducted on a Linux server equipped with an AMD Ryzen 9 7950X CPU (16 cores, 32 threads, 4.5 GHz), 128 GB of RAM, and a 2 TB NVMe SSD. Debun parses JavaScript files with the `meriyah`[5] parser, which supports the latest JavaScript syntax.

7.1 Data Collection

We collected JavaScript libraries from Cdnjs and npm,[6] and crawled bundled JavaScript files from high-traffic websites. We manually constructed a ground truth dataset indicating which libraries and their versions were used on each website.

7.1.1 Library Collection. We collected 78 libraries used in PTDetector comprising 8,256 versions. We excluded libraries unavailable on Cdnjs to enable a fair comparison with PTDetector, which relies on Cdnjs as its library source. We further excluded versions marked as pre-releases or nightly builds, as they are rarely used in practice. When multiple npm packages originate from the same GitHub repository, we treated them as a single library. For example, we treated `react` and `react-dom` collectively as React.

7.1.2 Website Collection. From the top 100 high-traffic websites listed by Semrush,[7] we selected 68 sites that could be loaded automatically from our crawling environment. Concretely, we discarded a site when our headless Puppeteer crawler could not reach its landing page (e.g., the request was blocked or geo-restricted from our location) or when the returned page exposed an anti-bot challenge such as a reCAPTCHA widget instead of the regular content. We detected both conditions automatically from the navigation response and the loaded DOM, without manual inspection. Using Puppeteer, we then crawled every JavaScript file loaded by these websites, rather than restricting collection to bundled outputs, so that the ground truth reflects all libraries shipped to end users regardless of how they are delivered (e.g., bundles, standalone CDN scripts, or directly served library files). All data was collected on 27 February 2025. This corpus also fixes the ubiquity filter of §3.2, as we remove from the library database every fingerprint that appears on more than 60% of these websites. We set this threshold above the prevalence of every library in the corpus. The most widespread one, core-js, appears on 54% of the websites, so a fingerprint that crosses 60% cannot be accounted for by any single library and is instead shared across libraries or injected by bundlers. The removed fingerprints bear this out, as each occurs in a median of seven different libraries in our database.

7.1.3 Ground Truth Collection. Since no fixed, up-to-date ground truth dataset exists for library and version detection, we manually constructed our own. We could not reuse PTDetector's ground truth dataset because live websites frequently update their dependencies, and the specific JavaScript files used during their evaluation are no longer available. We aggregated the detection results from PTDetector, LDC, and Debun using a conservative threshold ($\psi_u = 5\%$), and then manually verified whether each flagged library was actually present. Two authors independently performed this verification; when their labels disagreed, we accepted a label only if it was supported by explicit evidence in the code, such as an explicit library name, and discarded the entry otherwise. We labeled a library as present if the code contained library-specific identifiers or at least five distinct functions, each longer than ten lines. Distinguishing libraries was straightforward, but pinpointing exact versions was far harder: each library has an average of 106 candidate versions, and partially imported libraries or mixed version artifacts often

[5] https://www.npmjs.com/package/meriyah
[6] https://cdnjs.com/ and https://www.npmjs.com/
[7] https://www.semrush.com/website/top/

Just Accepted

obfuscate which one is present. We therefore assigned version annotations only to libraries whose versions could be unambiguously verified through explicit code indicators (e.g., `_.version = "4.17.21"`), license texts, or developer comments. Furthermore, because the databases for D-Bundlr and Aletheia are built strictly from npm, we excluded library versions not published on npm from our ground truth to ensure a fair comparison. The resulting ground truth dataset [25] contains 223 verified library instances, that is, pairs of a website and a library present on it, spanning 31 distinct libraries. Of these instances, 98 carry explicit version annotations.

7.2 Baseline Selection and Configuration

We select baseline tools by two criteria: 1) they represent the state of the art in library or version detection on the web, and 2) they provide publicly available artifacts executable on our dataset. For library detection (§7.5), we compare Debun against LDC [1], a widely used open-source detector; PTdetector [18], a property pattern-based detector that automatically collects the patterns LDC configures manually; and D-Bundlr [38], the most recent detector specifically targeting bundled libraries. For version detection (§7.6), we compare against LDC, Aletheia [31], and PTV [17], an extension of PTdetector to version detection. PTdetector and D-Bundlr themselves support only library detection. We exclude URR [3] because its exhaustive hashing does not scale to our library database (§9).

These tools fall into two groups that we distinguish throughout the evaluation. LDC, PTdetector, and PTV are *runtime* detectors that inspect the objects a page materializes once it is executed in a browser, whereas D-Bundlr, Aletheia, and Debun are *static* tools that analyze the delivered source without running it. We ran LDC and PTdetector on the live pages during the crawl of §7.1, so their detections correspond to the same page states as the archived files. The remaining three baselines (D-Bundlr, Aletheia, and PTV) all postdate that crawl. Being static, D-Bundlr and Aletheia analyze the archived files directly, whereas PTV requires executing the target website, which our archived files do not allow. We therefore exclude PTV from the website evaluation and instead compare against it on a controlled single-library corpus at the end of §7.6.

LDC and PTdetector. We run LDC's `js-library-detector` npm package at its latest version, 6.7.0, without modification. We evaluate PTdetector using its default settings: a score threshold of 0.5 and a depth limit of 3.

D-Bundlr. Because D-Bundlr expects each input to expose a Webpack-style container that maps module IDs to module functions, we hand it the largest function-bearing array, object, or statement in each file as that container. This covers at least 80% of the input for 78% of the files, with a median of 106 modules extracted, so D-Bundlr is not penalized by format mismatches. We train its classification model on *our* library set with the pipeline of its original publication [38] and no manual intervention. Preprocessing succeeded for 21 of the 31 ground-truth libraries, while the other 10 failed automatic bundling (7), yielded no features (2), or timed out (1), reflecting that the pipeline presupposes automatically bundlable npm artifacts. Since a supervised detector can only emit libraries from its training set, we score D-Bundlr over those 21 only (159 of the 223 ground-truth instances), and because this scope differs from that of the other detectors, we compare it with Debun separately over the same 21 libraries (§7.5). Its dynamic validation phase could not be applied at all, as discussed in §7.5.

Aletheia. Aletheia requires the target library to be given as input and reports only which of its versions are present, which is why it appears only in the version comparison (§7.6). Accordingly, only its compartment-less mode could be applied to our dataset.

PTV. Since PTV extends PTdetector, we evaluate it for version detection on a controlled single-library corpus (§7.6). We run PTV unmodified, at version 1.5 of its published extension, under the detection parameters its own entry point sets: a property-tree depth limit of 2, a size limit of 10,000 visited nodes, and a 5s cap on its traversal.

Just Accepted

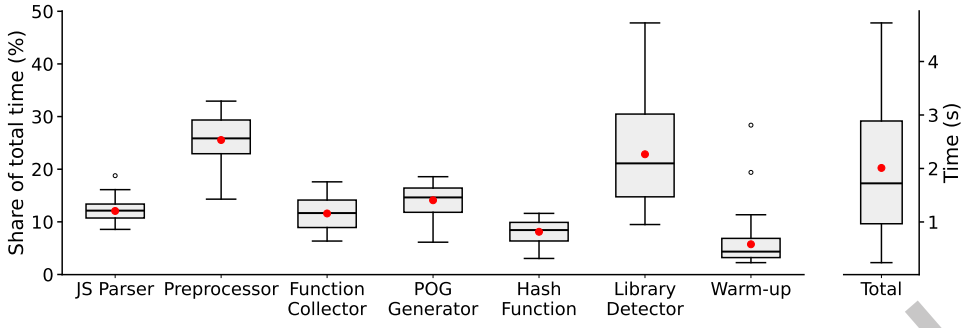


Fig. 14. Share of the per-website detection time spent in each step of Debun (left) and the distribution of the total time (right).

Reproducibility. The exact tool versions, configurations, adaptation code, and scripts for running every baseline on our dataset are included in our replication package.

7.3 RQ1. Performance

Throughout this subsection, Debun runs in the configuration that §7.4 identifies as best performing, so that the cost we report is the cost of the configuration evaluated in §7.5 and §7.6. Debun takes 30 minutes to construct the library database, consuming 39.07 MB of storage space. This is a one-time cost, and the rest of this section reports the recurring cost of analyzing a website. We report the recurring cost on the 50 websites among the top 100 on which all six tools run, so that every number in this section, including the comparison in Table 1, comes from the same input set. Since the runtime detectors cannot be timed on archived files, we measured this cost by running all six tools together on pages crawled in October 2026, separately from the archived corpus of §7.1 on which the detection results are computed. Because runtime measurements are sensitive to the state of the machine, we repeated every measurement three times on the server described in §7 and report the mean over the three runs together with the standard deviation across them. The one exception is the crawl, which we performed once so that every run of the static tools analyzes the same files. Figure 14 shows, for each website, the share of Debun's detection time spent in each step, together with the distribution of the total time. The fingerprint collection stages (JavaScript parsing, preprocessing, function collection, POG generation, and hashing) account for 71% of the per-website time on average, and the library detector for 23%. Within fingerprint collection, the preprocessing (function inlining) pass is the most expensive stage at 26%, as it performs the def-use analysis and source-to-source rewriting over every collected function. The remaining 6% (80 ms per website), shown as *Warm-up*, is the start-up of the Node.js worker in which each website is analyzed. Even so, Debun processes each website in 2.0 s on average, which is practical for large-scale analysis. In terms of memory, the 39.07 MB library database expands to 684 MB once parsed into the in-memory fingerprint index. Since this index is built once and shared across all detections, analyzing a single website uses 1.22 GB of memory on average and 2.20 GB at worst.

Table 1 compares this cost with that of the baseline tools described in §7.2. We restrict the site set this way because the runtime detectors cannot be applied to websites that fail to load in their browser environment, so only on this subset does every tool analyze the same input under the same conditions. PTV appears here for its cost only, as its detections cannot be aligned with our archived corpus. Besides the analysis itself, each tool must first make a website analyzable, and this setup cost differs by design. Debun and the other static tools crawl the scripts a website serves, which takes 20.09 s, whereas the runtime detectors spend 1.6–1.8 s loading and executing the page in a browser. Both costs are incurred once per website, so the crawl is the more expensive setup, and it

Just Accepted

Table 1. Detection cost per website in seconds, averaged over the 50 websites among the top 100 on which all six tools run, as the mean ± standard deviation over three runs. *Setup* is a browser page load for the runtime tools and a crawl of the served scripts for the static tools, which share one crawl.

Metric	**Runtime tools**			**Static tools**		
	LDC	PTDetector	PTV	D-Bundlr	Aletheia	Debun
Setup	1.74±0.06	1.81±0.03	1.65±0.05	20.09	20.09	20.09
Detection	0.03±0.00	0.06±0.00	0.24±0.00	2,571.09±24.05	235.61±2.39	2.01±0.03
Total	1.76±0.06	1.86±0.04	1.89±0.05	2,591.18±24.05	255.70±2.39	22.10±0.03

is only amortized when the crawled scripts are analyzed more than once, as they are here by the three static tools that share one crawl and as they would be whenever the fingerprint database is updated and detection is re-run. A single analysis of one website therefore costs Debun 22.10 s against 1.8–1.9 s for the runtime detectors, an order of magnitude more, almost all of it the crawl. Once a page is loaded, the runtime detectors inspect the objects it has already materialized and finish in under 0.25 s, whereas Debun reconstructs POGs from the source and is therefore one to two orders of magnitude slower. The learning-based detectors are slower still, by two to three orders of magnitude, taking 2,571 s and 236 s per website, because both invoke a neural model on every extracted module and the websites in this subset contain 5,847 modules on average. The standard deviations across the three runs are small relative to the differences between tools (Table 1), so none of these comparisons changes from one run to another.

Answer to RQ1

Debun detects libraries in 2.0 s per website on average, using at most 2.20 GB of peak memory, most of which is the shared fingerprint index. Adding the 20.09 s crawl, it is an order of magnitude slower end to end than the runtime detectors and one to two orders of magnitude faster than the learning-based ones.

7.4 RQ2. Ablation Study

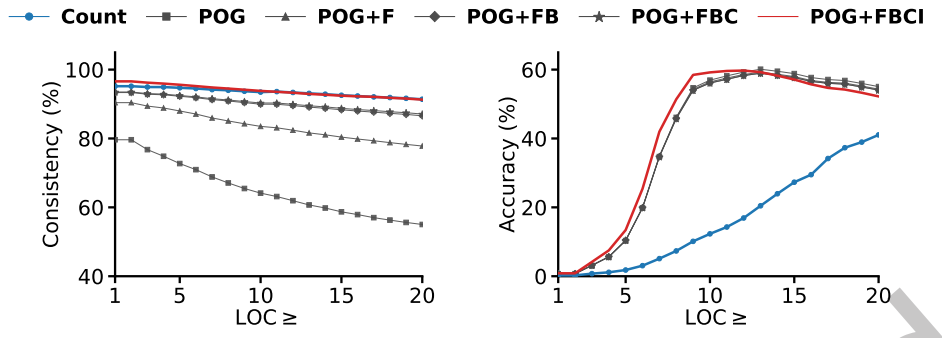
To understand the contribution of each design choice in Debun, we conduct two ablation studies: one on the POG generation techniques and another on the library scoring function.

7.4.1 POG Ablation. We first evaluate the effectiveness of the POG-based function fingerprints by comparing different POG generation techniques:

- Count – Count of each property operation without execution order.
- POG – POGs via basic construction algorithm (§4.1).
- POG+F – POG with *branch flipping*.
- POG+FB – POG+F with *branch bypassing*.
- POG+FBC – POG+FB with *path cloning*.
- POG+FBCI – POG+FBC with *function inlining*.

The four techniques are independent of one another, and each can be enabled on its own. We add them cumulatively, in the order in which Debun applies them, because the consistency curves of the models in Figure 15a then stack in order: each model differs from its predecessor in exactly one technique, so the gap between two adjacent curves shows the contribution of that technique. For each technique, we compute two metrics: *consistency*, the

Just Accepted



(a) Consistency (%) (b) Accuracy (%)

Fig. 15. Scores for each fingerprinting models with LOC $\geq x$.

proportion of functions whose fingerprints remain unchanged both before and after transpilation, and *accuracy*, which indicates how well fingerprints differentiate between different functions. Accuracy compares the function count against # Duplicated, a count of the fingerprint collisions among the collected functions, so it is higher when fewer functions share a fingerprint.

$$\text{Consistency} = \frac{(\#\text{ Consistent})}{(\#\text{ Functions})} \qquad \text{and} \qquad \text{Accuracy} = \frac{(\#\text{ Functions})}{(\#\text{ Duplicated})}$$

We collect 256,884 function hashes from the latest versions of all 78 libraries collected in §7.1. To reduce potential bias, we remove functions with identical syntax, which yields a final dataset of 91,898 functions for Count and the POG-based models from POG to POG+FBC. For POG+FBCI, function inlining further merges small callees into their callers, so the function count is smaller; we compute its consistency and accuracy over this reduced set. We then transpile them using Terser and SWC with the most aggressive minify options except for unsafe options. We select them because they are the minifiers that mainstream production build pipelines use by default. Figure 15 compares model scores across different line-of-code (LOC) thresholds x. To avoid the bias from the original source formatting, we measure LOC after regenerating each function with Escodegen [30], a standard ECMAScript code generator, using its default formatting options. This yields a uniform, formatter-independent LOC for every function regardless of its original whitespace or style. As a representative operating point, Table 2 reports the per-model function and duplicate counts together with the resulting consistency and accuracy at LOC ≥ 6.

Consistency. While Count might appear consistent at first glance since it simply counts the number of property operations, function inlining introduces additional property operations and changes the count, breaking its consistency. In contrast, our POG-based models resolve this issue via the function inlining preprocessing pass. Consequently, POG+FBCI fundamentally distinguishes itself from Count and achieves the highest consistency among all models (95.21% versus 94.51% for Count at LOC ≥ 6 in Table 2). The baseline POG model exhibits the lowest consistency (70.95%) due to the inconsistency of the control flow explained in §2.2. As Table 2 shows, the consistency improves monotonically as control-flow refinement techniques are progressively applied (70.95% → 87.10% → 91.84% → 92.08%), and the branch flipping has the greatest impact.

Accuracy. All POG-based models achieve significantly higher accuracy than Count, which performs the worst, highlighting the importance of preserving the execution order of property operations for distinguishing functions after transpilation. Among the POG-based models, the three control-flow refinements leave accuracy essentially

Just Accepted

Table 2. Scores for each model when LOC ≥ 6.

Metric	Count	POG	POG+F	POG+FB	POG+FBC	POG+FBCI
# Consistent	52,469	39,387	48,354	50,985	51,121	48,643
# Functions	55,517	55,517	55,517	55,517	55,517	51,091
Consistency	94.51%	70.95%	87.10%	91.84%	92.08%	**95.21%**
# Functions	55,517	55,517	55,517	55,517	55,517	51,091
# Duplicated	1,812,355	279,897	278,871	278,911	278,917	201,245
Accuracy	3.06%	19.83%	19.91%	19.90%	19.90%	**25.39%**

Table 3. End-to-end library detection (top) and version detection (bottom: # correctly detected versions out of 98 with average candidate-range sizes) for each POG generation technique with the module-dependency-aware scorer at $\psi_m = 84\%$.

Metric	Count	POG	POG+F	POG+FB	POG+FBC	POG+FBCI
Precision	94.84%	97.13%	96.65%	96.65%	96.65%	**98.54%**
Recall	90.58%	75.78%	90.58%	90.58%	90.58%	**91.03%**
F1-score	92.66%	85.14%	93.52%	93.52%	93.52%	**94.64%**
Major	91 (1.06)	73 (1.05)	92 (1.01)	92 (1.01)	92 (1.01)	**94 (1.02)**
Minor	89 (1.77)	65 (1.86)	87 (1.57)	88 (1.56)	88 (1.56)	**91 (1.73)**
Patch	82 (3.55)	57 (3.35)	80 (2.96)	82 (2.94)	82 (2.95)	**88 (3.25)**

unchanged (POG 19.83% to POG+FBC 19.90%), as they normalize how a fingerprint is computed without making distinct functions collide. Function inlining raises it to 25.39%, since merging a single-use callee into its caller yields a larger POG that collides with fewer other functions. Part of this gain also reflects the smaller function set that inlining leaves behind, as noted above, so the two columns are not measured over identical populations.

End-to-End Impact. Consistency and accuracy isolate the quality of the fingerprints themselves, independently of the scoring function and threshold. We also connect them to the final detection quality. Table 3 reports end-to-end library and version detection for each POG generation technique on the evaluation dataset, using the module-dependency-aware scorer at $\psi_m = 84\%$. The basic POG model performs worst (85.14% F1): its low consistency means that too few of its fingerprints survive bundler transformations, dropping recall to 75.78%. Branch flipping recovers the lost recall (90.58%), confirming that the consistency improvements in Table 2 translate into end-to-end gains. The effect of branch bypassing and path cloning does not show up in library detection on our dataset, as POG+F, POG+FB, and POG+FBC all reach 93.52% F1. Both target transformations that minifiers commonly apply, namely merging nested branches into short-circuit logical expressions and merging duplicated branch bodies into a shared block (§4.3), and they raise fingerprint consistency accordingly (87.10% → 91.84% → 92.08% in Table 2). In our website corpus, however, the functions they affect rarely decide whether a library clears the threshold, so this gain is not visible in library detection, and only version detection improves slightly with bypassing. We retain both techniques because they cost no accuracy and keep the fingerprints stable under these minifier transformations wherever they occur. Function inlining improves precision and recall together

Just Accepted

Table 4. Detection performance across different similarity scoring functions: 1) uniform score score_u and 2) module-dependency-aware score score_m. Each scorer is reported at its own best-F1 threshold ($\psi_u = 15\%$ for score_u and $\psi_m = 84\%$ for score_m). Parenthesized values are average candidate-range sizes.

(a) Library detection.

Metric	score_u	score_m
TP	202	203
FP	16	3
FN	21	20
Precision	92.66%	**98.54%**
Recall	90.58%	**91.03%**
F1-score	91.61%	**94.64%**

(b) Version detection.

Level	score_u	score_m
Major	94 (1.01)	94 (1.02)
Minor	89 (1.45)	91 (1.73)
Patch	84 (2.18)	88 (3.25)

(c) Detected instances per library, listing the six most frequent ones and aggregating the rest as *Others*.

Library	score_u	score_m	Ground
React	**35**	34	35
core-js	26	**37**	37
Lodash.js	27	**32**	34
jQuery	30	30	30
Preact	**10**	5	10
Zepto	10	10	10
Others	64	55	67
Total	**202**	**203**	**223**

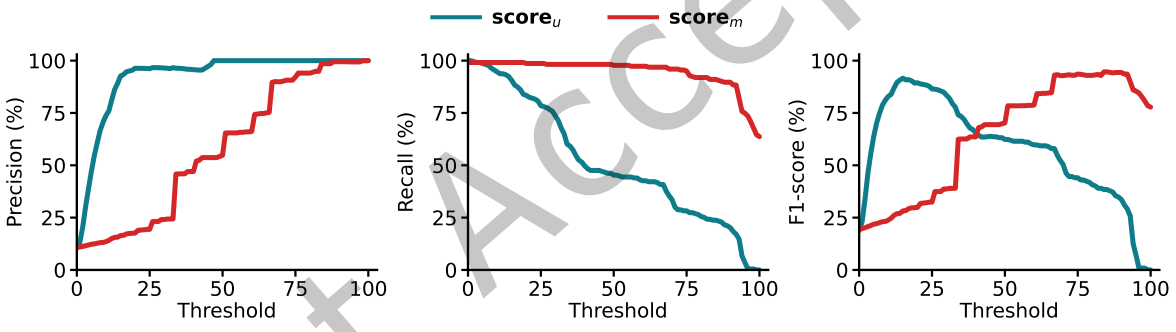


Fig. 16. Library detection scores across different fingerprinting models and library scorers.

(96.65% → 98.54% and 90.58% → 91.03%), yielding the best overall F1-score (93.52% → 94.64%). The four spurious matches it removes are all partial matches against Lodash.js, where a few generic utility functions had matched a library the website does not actually ship. It also recovers four true matches and loses three, and the losses are an artifact of the threshold rather than of inlining itself. They occur on libraries of which the website imports only a handful of functions, so inlining merges those few into their callers and leaves so small a fingerprint set that a single unmatched entry drops the match ratio below $\psi_m = 84\%$. Version detection follows the same trend, with the full model correctly identifying the most versions at every semantic-versioning level.

From the evaluation results, we show that the POG-based function fingerprints are effective in representing the function with high consistency and accuracy. Each CFG refinement step improves consistency while preserving accuracy. In library detection, low consistency results in missing matches, thereby reducing recall, while low accuracy leads to incorrect matches, decreasing precision but potentially increasing recall. This connection is

Just Accepted

directly visible in Table 3: the least consistent model (POG, 70.95% consistency) loses the most recall at 75.78%, and branch flipping, which brings the largest consistency gain, also recovers the most end-to-end recall. The opposite direction appears in Count, whose consistency is high (94.51%) but whose accuracy is only 3.06% because a bare count of property operations collides across unrelated functions, and which accordingly records the lowest precision of the six models at 94.84%. Only POG+FBCI is best on both intermediate metrics (95.21% consistency and 25.39% accuracy), and it is also the only model to reach the best F1-score.

7.4.2 Library Scorer Ablation. We next compare different library scorers on actual bundled JavaScript detection tasks to study how scoring affects detection quality. We denote the variant of Debun using the uniform scorer as score_u and the variant using the module-dependency-aware scorer as score_m. We report each scorer at its own best-F1 operating point, $\psi_u = 15\%$ for score_u and $\psi_m = 84\%$ for score_m. Table 4a shows that the dependency-aware scorer improves library detection F1-score from 91.61% (score_u) to 94.64% (score_m), with higher precision (92.66% → 98.54%) and slightly higher recall (90.58% → 91.03%). Figure 16 sweeps the threshold for both scorers across all fingerprinting models and shows why. Without dependency information, tree shaking quickly erodes recall: as the score threshold is raised above 15%, score_u's recall drops sharply because each library no longer retains enough surviving fingerprints to clear the threshold. In contrast, score_m maintains a high recall even at large thresholds, since the dependency-aware scorer aggregates evidence across related modules and compensates for fingerprints removed by tree shaking. Table 4b shows the same trend for version detection at the three semantic-versioning levels (§7.6), where score_m matches score_u at the major level and detects more versions at the minor and patch levels (94, 91, 88 versus 94, 89, 84 at major, minor, and patch, respectively). Table 4c further breaks down detection by library: score_m recovers more core-js (37 vs. 26) and Lodash.js (32 vs. 27) instances, which score_u missed because these libraries are typically partially imported and only a small subset of their fingerprints survive in the bundle, falling below the score threshold. In the other direction, score_m trails score_u on Preact (5 versus 10) and on the aggregated *Others* (55 versus 64), which is why the two scorers end up with nearly the same total. For Preact, this occurs because it ships only its built `dist` bundle on CDNs; thus, at high thresholds, the dependency-aware scorer cannot extract enough dependency information from npm to compensate. The exchange is nevertheless worth making, since it removes 13 of the 16 false positives of score_u (Table 4a) and raises the F1-score, whereas the two scorers detect almost the same number of instances in total. These results confirm that the choice of library scorer has a strong impact on detection performance.

Answer to RQ2

POG-based fingerprints become substantially more consistent as control-flow refinement techniques and function inlining are progressively applied (POG 70.95% → POG+FBCI 95.21%). The refinements leave accuracy unchanged and inlining improves it (19.83% → 25.39%), and the module-dependency-aware scorer further improves library detection F1-score from 91.61% to 94.64%.

Based on these results, we answer RQ3 and RQ4 by evaluating Debun with the best-performing configuration identified in this ablation study: POG fingerprints with all control-flow refinements and function inlining (POG+FBCI), combined with the module-dependency-aware scorer (score_m) at $\psi_m = 84\%$, the 60% website-frequency filter (§7.1), and the 30% library-overlap filter (§6.1).

7.5 RQ3. Library Detection

We compare the effectiveness of Debun against state-of-the-art library detection tools: LDC, PTdetector, and D-Bundlr, selected and configured as described in §7.2. LDC and PTdetector are scored over all 223 ground-truth instances, whereas D-Bundlr can only be scored over the 21 libraries on which its classifier could be trained.

Just Accepted

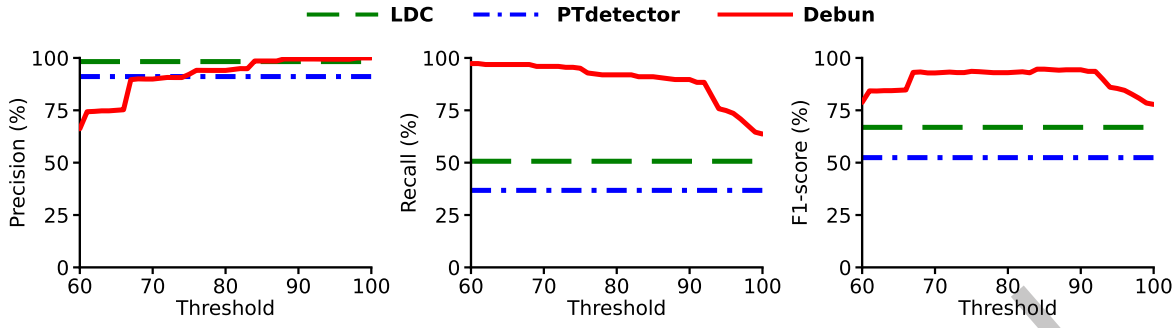


Fig. 17. Library detection scores with different thresholds ψ_m.

Table 5. Library detection over all 223 ground-truth instances ($\psi_m = 84\%$).

Metric	LDC	PTdetector	Debun
TP	113	82	203
FP	2	8	3
FN	110	141	20
Precision	98.26%	91.11%	**98.54%**
Recall	50.67%	36.77%	**91.03%**
F1-score	66.86%	52.40%	**94.64%**

Table 6. Library detection over the 21 libraries of D-Bundlr (159 instances).

Metric	D-Bundlr	Debun
TP	100	147
FP	401	0
FN	59	12
Precision	19.96%	**100.00%**
Recall	62.89%	**92.45%**
F1-score	30.30%	**96.08%**

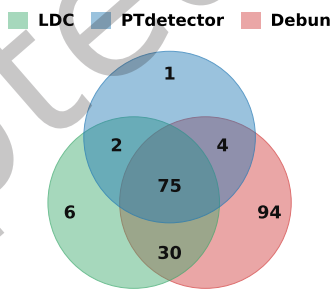


Fig. 18. Library instances detected in Table 5.

To keep every comparison within one evaluation scope, Figures 17 and 18 and Table 5 compare Debun with LDC and PTdetector, and Table 6 compares Debun with D-Bundlr separately over the libraries D-Bundlr covers.

Score Threshold Sensitivity. To study the sensitivity of Debun to the score threshold ψ_m, we sweep ψ_m from 60% to 100% in 1% increments and report precision, recall, and F1 in Figure 17. We omit true negatives (TN), as the libraries absent from an input are far too numerous. Debun obtains a higher F1-score and a higher recall than LDC and PTdetector at *every* threshold. For precision, Debun exceeds PTdetector for $\psi_m \geq 75\%$ and LDC for $\psi_m \geq 84\%$. Hence the conclusion that Debun outperforms the baselines does not hinge on any particular threshold value. For a single, comparable headline number, Table 5 reports Debun at the threshold that yields its best F1 on this dataset, $\psi_m = 84\%$ (94.64%), which is 1.42× and 1.81× higher than those of LDC and PTdetector, respectively.

Recall. Figure 18 shows the Venn diagram of library instances detected by each tool. Debun detects 90 and 121 more library instances than LDC and PTdetector, respectively, because they cannot detect libraries whose top-level properties are mangled (e.g., React) or not exported to the global object (e.g., Lodash.js). For example, Debun detected both React and Lodash.js in pinterest.com, a design resource website, but LDC and PTdetector failed to detect them. Debun's remaining 20 false negatives have two causes. First, too few fingerprints of a library survived bundler transformations to clear the score threshold, so the MDG could not be reliably constructed and the library fell out of the candidate entry set. Second, some library versions appear on CDNs but were never

Just Accepted

published to npm, so Debun cannot retrieve their module structure, and the version is missed even when its fingerprints are present in the bundle.

Precision. Debun achieves the highest precision, narrowly surpassing LDC, which itself benefits from various manual ad-hoc calculations. PTdetector struggles to distinguish libraries with overlapping property patterns. For example, Lodash.js and Underscore.js share a similar property pattern, _, which leads PTdetector to confuse the two and produce incorrect detections. This suggests that property patterns are not sufficiently distinguishable. While we mitigate the overlapping function issue by applying code segmentation, Debun still faces three false positives, all from shrink-wrapped libraries: the libraries that they copy from are not included in our database, so the filtering step cannot remove the borrowed functions.

Comparison with D-Bundlr. Table 6 compares Debun with D-Bundlr over the 21 libraries on which D-Bundlr's classifier could be trained, the only scope over which D-Bundlr can be scored (§7.2). Within this scope, Debun reaches a 96.08% F1-score against 30.30% for D-Bundlr, with both a higher recall (92.45% versus 62.89%) and a higher precision (100.00% versus 19.96%). D-Bundlr can only detect libraries that are actually bundled, so it misses libraries loaded externally (e.g., via CDN), and it frequently misses Lodash.js and core-js, whose unused functions tree shaking eliminates. Its low precision partly reflects its dynamic validation stage, which filters the classifier's false positives by executing the candidate modules but failed on every input, as it assumes Webpack-style module boundaries that the general web artifacts in our dataset do not exhibit. A successful validation would have raised only its precision, not its recall.

Answer to RQ3

Debun outperforms LDC and PTdetector in library detection, achieving both a higher F1-score and a higher recall at every threshold, and its F1-score at the representative operating point (94.64%) is 1.42× and 1.81× higher than theirs. Over the libraries D-Bundlr can be trained on, Debun also outperforms D-Bundlr (96.08% versus 30.30% F1-score).

7.6 RQ4. Library Version Detection

While different libraries have distinct fingerprints, versions of the same library often differ only slightly. Tree shaking may remove version-specific functions, making them harder to distinguish, and libraries may not follow strict semantic versioning; for example, Lodash.js v4.17.14 and v4.17.15 differ only in version labels without any code changes. For this reason, Debun and Aletheia output a range of candidate versions rather than a single version: we cannot verify whether two versions truly have no function-level differences or simply differ in ways that the tools could not detect. LDC, on the other hand, typically reads the version label and emits exactly one version, but when an explicit label is missing, it falls back to analyzing version-specific code characteristics and may emit only the major or minor portion of the version. We therefore evaluate library version detection at three semantic-versioning levels: *Major*, *Minor*, and *Patch*. For each level, we report the number of correctly detected versions and the average size of the candidate range returned by each tool, where a smaller range indicates more precise version identification. On the website dataset we compare Debun with LDC and Aletheia, selected and configured as described in §7.2, and we compare against PTV on a controlled corpus at the end of this subsection.

Table 7 compares the version detection effectiveness of LDC, Aletheia, and Debun on the 98 version-annotated ground-truth instances of §7.1. LDC misses 47 of them outright because it fails to detect the underlying library at all, whereas Debun fails at the library level for only 2. On top of this library-level gap, LDC also struggles at the version level: it detects library versions only if a version label exists in the code, and breaks when libraries have no explicit version labels or use inconsistent labels across versions. For example, the version label for core-js is `core.version` before v0.9.12, but it was changed to `__core-js-shared__.version` in v0.9.12. Together,

Just Accepted

Table 7. Library version detection at $\psi_m = 84\%$. Each cell: # correct versions out of 98 and the average candidate-range size.

Level	LDC	Aletheia	Debun
Major	41 (1.00)	88 (1.03)	94 (1.02)
Minor	41 (1.61)	75 (1.31)	91 (1.73)
Patch	41 (3.56)	67 (2.74)	88 (3.25)

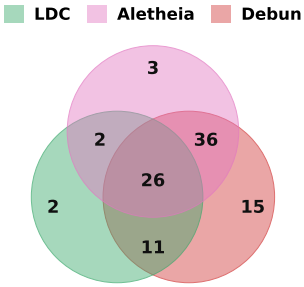


Fig. 19. Venn diagram of versions correctly detected at the patch level.

Table 8. Library version detection of PTV and Debun on the controlled single-library corpus (45 libraries, 2,113 library–version pairs). Each cell is the fraction of pairs whose reported candidate versions contain the ground-truth version, with the average candidate-range size in parentheses.

Level	PTV	Debun
Major	73.9% (1.06)	**86.7% (1.08)**
Minor	65.8% (1.26)	**81.3% (1.27)**
Patch	63.8% (2.23)	**76.5% (2.13)**

these issues result in many false negatives in LDC across all three levels, where it detects only 41 versions regardless of granularity. LDC's average range size grows from 1.00 at the major level to 3.56 at the patch level: LDC reads a version label and emits a single version, but for libraries such as Preact, the label only identifies the major version, so a single emitted label covers multiple minor and patch versions, widening the candidate range at finer granularities. Aletheia detects more versions than LDC (88 at major) but degrades sharply at finer granularities, dropping to 75 at minor and 67 at patch. In many of these patch-level misclassifications, the score of the ground-truth version was slightly below the score of the detected version, indicating a degrading effect of the noise from extraneous code inside the input. In contrast, Debun correctly detects 94, 91, and 88 versions at the major, minor, and patch levels, respectively, outperforming both baselines at every level with only a small drop across granularities. Although Debun's candidate range at the patch level (3.25) is slightly larger than Aletheia's (2.74), this trade-off comes from Debun identifying many more versions that Aletheia misses entirely, as also visualized in Figure 19. This demonstrates that POG provides high accuracy for version-unique or library-unique fingerprints in real-world scenarios, enabling accurate version identification without explicit version labels. The two tools thus serve complementary use cases, although the difference in range size is modest: Aletheia suits scenarios that demand a precise version with a tighter candidate range for an already-identified library, whereas Debun suits scenarios where no library or version may be missed, at the cost of a slightly wider candidate range. When both are applicable, they can also be combined as a pipeline: Debun first detects the libraries and their candidate versions end-to-end, and Aletheia, given each detected library as its input, further narrows the candidate range.

Comparison with PTV. Although PTV cannot be run on our archived dataset (§7.2), we additionally compare version detection in a controlled setting favorable to PTV: following the single-library protocol of its own evaluation, we load each library version's canonical cdnjs distribution file into an otherwise empty page, removing

Just Accepted

bundling from the equation. Both tools receive the identical bytes: PTV executes the file in a headless browser and inspects the `window` object as shipped, while Debun statically fingerprints the same file with the module-dependency-aware scorer at $\psi_m = 84\%$, the same configuration as in Table 7. The corpus is the intersection of the two tools' version databases, 45 libraries and 2,113 library–version pairs. PTV is additionally given best-effort outer dependencies (e.g., jQuery for jquery-mobile) so that dependency-only load failures do not understate its results. Table 8 shows that Debun correctly identifies more versions at every semantic-versioning level (76.5% versus 63.8% at the patch level) with a tighter candidate range at the patch level. The two tools fail on different inputs. PTV reads properties off the `window` object, so it misses libraries that never attach themselves to the global scope and those that do but under mangled names, the limitation of property-based detection discussed in §2.1. Debun instead loses precision on very small libraries, which expose too few functions to yield fingerprints that separate one version from its neighbors, so it reports the right library with an imprecise version. Since this protocol removes bundling, these results characterize the controlled single-library setting and should not be interpreted as a direct estimate of PTV's performance on real bundled websites.

Answer to RQ4

Debun correctly identifies 94, 91, and 88 of the 98 ground-truth versions at the major, minor, and patch levels, respectively, outperforming both LDC and Aletheia at every level with a comparable candidate-range size. It also identifies more versions than PTV at every level on the controlled single-library corpus, where bundling is removed in PTV's favor.

8 Discussion

8.1 Advantages and Limitations

8.1.1 Advantages of Debun and POG. Debun and its POG-based fingerprints offer three main advantages. First, Debun achieves up to 2.5 times the recall of state-of-the-art tools while maintaining the highest precision among them, demonstrating the effectiveness of POGs as robust fingerprints for bundled library detection. Second, Debun is a fully automated, static-analysis-based approach that eliminates the manual effort required to add new libraries or versions. This contrasts with PTDetector, which relies on dynamic execution and so needs each library version to load in a browser with the outer dependencies we supplied by hand (§7.6). Third, POG-based fingerprints apply beyond library detection to any task that must identify original code inside heavily transpiled bundles, such as vulnerable function detection, code provenance analysis, and license compliance checking.

8.1.2 Limitations of Debun and POG. During the path-sensitive truthiness analysis (§4), we use the number of flipped and non-flipped values to determine branch flipping. In edge cases where these counts are exactly equal, our approach cannot deterministically resolve the flip direction. It then picks one direction, so the two sides of a comparison may normalize differently and the function fails to match.

Our normalization also covers only the transformations we model. Function inlining fires where the def-use analysis can establish its premises and skips the call site otherwise (§5.1), and the flow refinement normalizes the three rewriting patterns of §4.3. A bundler that inlines at several call sites at once, or that rewrites control flow in some other way, therefore produces code we do not bring to the same normal form. The cost of these cases falls on recall rather than precision, since a function we fail to normalize does not match a wrong library but simply fails to match, and the 95.21% consistency under the most aggressive non-unsafe Terser and SWC options (§7.4.1) bounds how often they arise.

In addition, Debun analyzes the JavaScript files delivered when a page is loaded: our crawler collects every script fetched at load time, but code that is dynamically loaded only after user interactions (e.g., lazily loaded route chunks fetched on navigation or click) is not collected and thus escapes detection. Combining Debun with

Just Accepted

interaction-driven crawling techniques [21] to cover such lazily loaded scripts is an interesting direction for future work.

Debun also assumes that each library appears in a single version per website, whereas one site can in fact use several versions of it. On amazon.com, for instance, a comment indicated jQuery v1.6.4, but certain functions had been upgraded to a newer version. Our evaluation does not account for such version heterogeneity. Since different versions typically arrive in different bundles, separating the individual bundles within a site and detecting versions per bundle could address this case, which we leave as future work.

Finally, POG rests on the premise that property names and the execution order of property operations are preserved after bundling (§2.2). Obfuscators that rewrite property names or turn property accesses into indirect ones (e.g., the advanced optimization mode of the Google Closure Compiler) break this premise, and Debun may miss libraries processed by such tools. However, renaming externally observable properties is unsafe for code that interacts with other scripts, so mainstream bundlers and minifiers preserve property names by default; indeed, POG fingerprints remain about 95% consistent under the Terser and SWC options measured above. Thus, POG's premise held under the evaluated Terser and SWC configurations, while property-renaming obfuscators remain outside our supported scope.

8.2 Threats to Validity

8.2.1 Construct Validity. Our ablation study measures fingerprint quality with two intermediate metrics, consistency and accuracy, which are only proxies for detection quality: a fingerprint set can be self-consistent yet still fail to identify libraries once a scoring function and a threshold are applied. We therefore report end-to-end detection for every POG generation technique as well (§7.4.1), where the least consistent model indeed loses the most recall. Measuring detection cost requires a construct as well, since making a website analyzable means crawling its scripts for the static tools but loading the page in a browser for the runtime ones, and the two differ by an order of magnitude. We therefore report setup and detection as separate rows rather than only a single total (§7.3). In addition, D-Bundlr is scored over the libraries its classifier could be trained on rather than the full library set, so its numbers describe a different population from the other detectors, and we therefore compare it with Debun only over that population (§7.5).

8.2.2 Internal Validity. Our study faces internal validity threats regarding the evaluation environment and ground truth construction. The moving target problem of live websites is a primary threat; websites frequently update their scripts, making it infeasible to reproduce the exact website states used in prior evaluations. While this mismatch may partly explain the performance drop of prior tools (which reported strong results on older datasets), we mitigated this by evaluating all tools simultaneously on freshly crawled data to ensure a strictly fair comparison. Additionally, the manual construction of our ground truth dataset may introduce human error. We minimized this risk by applying conservative thresholds and strictly verifying explicit code indicators (e.g., version labels, license texts) before confirming a match. A further threat concerns which candidates entered that manual verification: we seeded the pool with the detections of PTdetector, LDC, and Debun at a deliberately permissive threshold (§7.1), so a library that all three miss never enters the ground truth, and the recall we report for these three tools is an upper bound to that extent. D-Bundlr and Aletheia did not contribute to the pool, as we added them as baselines only in this revision. We therefore reviewed the detections they report outside our labels and found none that we should have labeled as present, so we do not expect their precision to be materially understated. Finally, we selected every threshold on the same corpus that we evaluate on, without a held-out split. The threshold sweep in §7.5 mitigates this for the score threshold, as Debun attains a higher F1-score and a higher recall than every baseline across the entire sweep rather than only at the tuned value. Furthermore, latent defects in our implementation could affect the reported results; we mitigate this threat by releasing our implementation together with all experiment scripts and raw results as a replication package, enabling independent verification.

Just Accepted

8.2.3 External Validity. External validity threats concern the generalizability of our results. The top 100 websites from Semrush and the selected libraries used for evaluation may not fully represent the vast diversity of all websites and JavaScript libraries in the wild. This also bounds the ubiquity filter of §7.1, which assumes that no single library reaches 60% of the analyzed websites. The assumption holds for our corpus, where the most widespread library appears on 54% of the websites, but not for the web at large, where jQuery alone runs on about 75% of pages [9]. Applied to such a corpus, the filter would discard jQuery's own fingerprints, so the threshold has to be chosen above the prevalence of the most common library in the corpus at hand.

8.2.4 Reliability. Reliability threats concern whether repeating our study would yield the same results. All datasets were collected on a fixed date, every threshold and configuration used in the evaluation is stated explicitly in the paper, and our replication package reproduces all reported numbers end-to-end from the raw corpora. The ground truth was labeled by two authors independently, and conflicting labels were kept only when supported by explicit evidence in the code (§7.1), reducing the dependence of our results on individual judgment.

9 Related Work

We review library detection for JavaScript and other domains, and code clone detection, and why bundling defeats each.

9.1 JavaScript Library Detection

Property pattern-based library detection techniques include LDC [1] and PTDetector [18]. While LDC requires manual configuration of property patterns, PTDetector can automatically collect them. PTV [17] further extends PTDetector to support version detection. However, these approaches become completely infeasible when runtime property patterns are modified or hidden during the bundling process. In particular, PTV cannot distinguish versions if their differences are confined to function body transformations. Furthermore, because these methods rely solely on exposed global objects rather than examining the underlying bundled code, their granularity in version detection is severely limited. URR [3] detects vulnerable JavaScript libraries by exhaustive hash-based matching. To account for all possible bundler transformations, URR generates a massive number of hashes, which suffers from severe scalability issues. It is designed to target a small, fixed set of libraries; reproducing its exhaustive hashing over our database of 78 libraries and 8,256 versions would require a prohibitively large fingerprint set, so we exclude URR from our evaluation. D-Bundlr [38] debundles bundled JavaScript files into their original modules and matches the recovered modules against a database, thereby improving static analysis to surface more security alerts. Aletheia [31] also performs hash-based matching to study the dependency update patterns of bundled JavaScript packages, but it requires the target library to be given as input and only identifies which versions of that library are bundled. Crucially, both D-Bundlr and Aletheia rely on AST-based hashing or similarity matching, which degrades when bundlers heavily alter the syntax. Our evaluation bears this out: D-Bundlr reaches only 19.96% precision on general web artifacts (§7.5), and Aletheia drops from 88 to 67 correctly identified versions as the granularity tightens from the major to the patch level (§7.6). The breadth of these approaches across vulnerability detection and dependency update studies highlights the critical importance of accurate JavaScript library and version detection.

9.2 Library Detection in Other Domains

Third-party library detection has been actively studied in other domains, such as iOS and Android apps. iOS approaches [8, 23] often rely on class-dumping or binary analysis, which are inapplicable to JavaScript libraries. Detection techniques for Android third-party libraries [4, 32, 35, 40, 42] use structural features such as class or method signatures and opcode sequences, making them resilient to traditional code obfuscation. Approaches like ATVHunter [39] extract fingerprints from control flow graphs (CFGs) to preserve execution order. However,

Just Accepted

applying these techniques directly to bundled JavaScript is ineffective; JavaScript minifiers actively alter control flows (e.g., via short-circuiting and branch flipping), necessitating the specialized flow-refinement and POG abstractions we introduce in Debun.

9.3 Code Clone Detection

Code clone detection techniques include text-based, token-based, tree-based, graph-based, and measure-based approaches. Text-based detection [11, 20], token-based detection [14, 28], and tree-based detection [41, 44] compare the similarity of source code text and syntax tokens. However, they are unsuitable for detecting bundled JavaScript libraries because minifiers radically alter the syntax. Graph-based detection techniques [43, 45] compare the similarity of structural graphs like control flow graphs or program dependency graphs. These approaches fundamentally assume that control flow and data flow remain largely preserved, which is an assumption that breaks under JavaScript bundling optimizations. Measure-based detection techniques [15, 27, 33, 37] calculate the similarity of extracted program features. While these techniques often exhibit higher resilience to syntax changes, they generally demonstrate lower precision, making them inadequate for pinpointing exact library versions in heavily bundled contexts.

10 Conclusion

In this paper, we presented Debun, a scalable approach for accurately detecting front-end JavaScript libraries and their versions in heavily bundled web applications. To overcome the limitations of existing techniques, we introduced the property-order graph (POG) to capture the invariant execution order of property operations. We further enhanced the robustness of POG-based fingerprints through a function-inlining preprocessing pass, and mitigated the effects of tree-shaking using a module-dependency-aware scoring function. Our extensive evaluation on real-world websites demonstrates that Debun significantly outperforms state-of-the-art tools (LDC, PTdetector, PTV, D-Bundlr, and Aletheia), achieving a 94.64% F1-score in library detection and correctly identifying 88 of the 98 ground-truth versions at the patch level. Ultimately, Debun provides a robust foundation for critical downstream tasks, such as automated vulnerability detection and software composition analysis.

Data Availability

The replication package for this paper is publicly available on GitHub [25]. It contains the implementation of Debun, the scripts that run every baseline under the configurations of §7.2, the library database and the crawled website corpus, the raw results behind every table and figure, and the ground-truth dataset together with the per-website evidence records that justify each label (§7.1).

Acknowledgments

This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No.RS-2024-00344597) and the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.RS-2024-00440780, Development of Automated SBOM and VEX Verification Technologies for Securing Software Supply Chains). The authors gratefully acknowledge the access to the Marvin cluster of the University of Bonn.

References

[1] 2023. js-library-detector: Library Detector for Chrome (LDC). https://www.npmjs.com/package/js-library-detector Version 6.7.0, published 27-Jul-2023.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

Just Accepted

[3] Mir Masood Ali, Peter Snyder, Chris Kanich, and Hamed Haddadi. 2024. Unbundle-Rewrite-Rebundle: Runtime Detection and Rewriting of Privacy-Harming Code in JavaScript Bundles. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (Salt Lake City, UT, USA) *(CCS '24)*. Association for Computing Machinery, New York, NY, USA, 2192–2206. doi:10.1145/3658644.3690262

[4] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and its Security Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 356–367. doi:10.1145/2976749.2978333

[5] SungGyeong Bae, Hyunghun Cho, Inho Lim, and Sukyoung Ryu. 2014. SAFEWAPI: Web API Misuse Detector for Web Applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 507–517. doi:10.1145/2635868.2635916

[6] 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 (POPL)*. doi:10.1145/512950.512973

[7] Patrick Cousot and Radhia Cousot. 1992. Abstract Interpretation Frameworks. *Journal of Logic and Computation (JLC)* 2, 4 (1992), 511–547. doi:10.1093/logcom/2.4.511

[8] Daniel Domínguez-Álvarez, Alejandro de la Cruz, Alessandra Gorla, and Juan Caballero. 2023. LibKit: Detecting Third-Party Libraries in iOS Apps. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1407–1418. doi:10.1145/3611643.3616344

[9] HTTP Archive. 2024. *The Web Almanac 2024: JavaScript*. https://almanac.httparchive.org/en/2024/javascript

[10] Seojin Kim, Sungmin Park, and Jihyeok Park. 2025. Debun: Detecting Bundled JavaScript Libraries on Web using Property-Order Graphs. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 78–90. doi:10.1109/ASE63991.2025.00015

[11] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. 595–614. doi:10.1109/SP.2017.62

[12] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2017. Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. In *Proceedings 2017 Network and Distributed System Security Symposium (NDSS 2017)*. Internet Society. doi:10.14722/ndss.2017.23414

[13] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. 2018. Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. *arXiv preprint arXiv:1811.00918*.

[14] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. CCLearner: A Deep Learning-Based Clone Detection Approach. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 249–260. doi:10.1109/ICSME.2017.46

[15] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. VulPecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (Los Angeles, California, USA) *(ACSAC '16)*. Association for Computing Machinery, New York, NY, USA, 201–213. doi:10.1145/2991079.2991102

[16] Ruyan Lin, Yulong Fu, Wei Yi, Jincheng Yang, Jin Cao, Zhiqiang Dong, Fei Xie, and Hui Li. 2024. Vulnerabilities and Security Patches Detection in OSS: A Survey. *ACM Comput. Surv.* 57, 1, Article 23 (Oct. 2024), 37 pages. doi:10.1145/3694782

[17] Xinyue Liu, Haipeng Cai, and Lukasz Ziarek. [n. d.]. PTV: Scalable Version Detection of Web Libraries and its Security Application. https://aaronxyliu.github.io/download/ptv-ready.pdf

[18] Xinyue Liu and Lukasz Ziarek. 2023. PTDETECTOR: An Automated JavaScript Front-end Library Detector. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 649–660. doi:10.1109/ASE56229.2023.00049

[19] Malavolta, Ivano and Nirghin, Kishan and Scoccia, Gian Luca and Romano, Simone and Lombardi, Salvatore and Scanniello, Giuseppe and Lago, Patricia. 2023. JavaScript Dead Code Identification, Elimination, and Empirical Assessment. *IEEE Transactions on Software Engineering* 49, 7, 3692–3714. doi:10.1109/TSE.2023.3267848

[20] George Mathew, Chris Parnin, and Kathryn T Stolee. 2020. SLACC: simion-based language agnostic code clones. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 210–221. doi:10.1145/3377811.3380407

[21] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. 2012. Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes. *ACM Transactions on the Web* 6, 1, Article 3 (2012), 30 pages. doi:10.1145/2109205.2109208

[22] Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. 2020. Detecting locations in JavaScript programs affected by breaking library changes. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 187 (Nov. 2020), 25 pages. doi:10.1145/3428255

[23] Orikogbo, Damilola and Büchler, Matthias and Egele, Manuel. 2016. CRiOS: Toward Large-Scale iOS Application Analysis. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices* (Vienna, Austria) *(SPSM '16)*. Association for Computing Machinery, New York, NY, USA, 33–42. doi:10.1145/2994459.2994473

[24] Jihyeok Park. 2014. JavaScript API Misuse Detection by Using TypeScript. In *Proceedings of the Companion Publication of the 13th International Conference on Modularity* (Lugano, Switzerland) *(MODULARITY 2014)*. Association for Computing Machinery, New York,

Just Accepted

NY, USA, 11–12. doi:10.1145/2584469.2584472

[25] Sungmin Park, Seojin Kim, Ben Swierzy, and Jihyeok Park. 2026. Replication Package for "Detecting JavaScript Libraries on the Web using Property-Order Graphs". https://github.com/ku-plrg/debun-tosem26

[26] Jeremy Rack and Cristian-Alexandru Staicu. 2023. Jack-in-the-box: An Empirical Study of JavaScript Bundling on the Web and its Security Implications. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 3198–3212. doi:10.1145/3576915.3623140

[27] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. 2018. Oreo: detection of clones in the twilight zone. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 354–365. doi:10.1145/3236024.3236026

[28] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 1157–1168. doi:10.1145/2884781.2884877

[29] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 361–376. https://www.usenix.org/conference/usenixsecurity18/presentation/staicu

[30] Yusuke Suzuki and contributors. 2025. Escodegen: ECMAScript code generator. https://github.com/estools/escodegen Accessed: 19-May-2026.

[31] Ben Swierzy, Marc Ohm, and Michael Meier. 2026. Insecure Ingredients? Exploring Dependency Update Patterns of Bundled JavaScript Packages on the Web. arXiv:2512.15447 [cs.SE] doi:10.1145/3744916.3787780

[32] Yan Wang, Haowei Wu, Hailong Zhang, and Atanas Rountev. 2018. ORLIS: obfuscation-resilient library detection for Android. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems* (Gothenburg, Sweden) *(MOBILESoft '18)*. Association for Computing Machinery, New York, NY, USA, 13–23. doi:10.1145/3197231.3197248

[33] Seunghoon Woo, Hyunji Hong, Eunjin Choi, and Heejo Lee. 2022. MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3037–3053. https://www.usenix.org/conference/usenixsecurity22/presentation/woo

[34] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. 2021. CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 860–872. doi:10.1109/ICSE43902.2021.00083

[35] Yafei Wu, Cong Sun, Dongrui Zeng, Gang Tan, Siqi Ma, and Peicheng Wang. 2023. LibScan: Towards More Precise Third-Party Library Identification for Android Applications. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 3385–3402. https://www.usenix.org/conference/usenixsecurity23/presentation/wu-yafei

[36] Elizabeth Wyss, Lorenzo De Carli, and Drew Davidson. 2022. What the fork? finding hidden code clones in npm. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2415–2426. doi:10.1145/3510003.3510168

[37] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, and Wenchang Shi. 2020. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1165–1182. https://www.usenix.org/conference/usenixsecurity20/presentation/xiao

[38] Wenyuan Xu, Alexi Turcotte, and Cristian-Alexandru Staicu. 2026. D-Bundlr: Destructing JavaScript Bundles for Effective Static Analysis. https://reallytg.github.io/files/papers/debundling_icse.pdf

[39] Xian Zhan, Lingling Fan, Sen Chen, Feng We, Tianming Liu, Xiapu Luo, and Yang Liu. 2021. ATVHunter: Reliable Version Detection of Third-Party Libraries for Vulnerability Identification in Android Applications. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1695–1707. doi:10.1109/ICSE43902.2021.00150

[40] Jiexin Zhang, Alastair R. Beresford, and Stephan A. Kollmann. 2019. LibID: reliable identification of obfuscated third-party Android libraries. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 55–65. doi:10.1145/3293882.3330563

[41] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 783–794. doi:10.1109/ICSE.2019.00086

[42] Yuan Zhang, Jiarun Dai, Xiaohan Zhang, Sirong Huang, Zhemin Yang, Min Yang, and Hao Chen. 2018. Detecting third-party libraries in Android applications with high precision and recall. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 141–152. doi:10.1109/SANER.2018.8330204

[43] Gang Zhao and Jeff Huang. 2018. DeepSim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 141–151. doi:10.1145/3236024.3236068

Just Accepted

[44] Deqing Zou, Hanchao Qi, Zhen Li, Song Wu, Hai Jin, Guozhong Sun, Sujuan Wang, and Yuyi Zhong. 2017. SCVD: A New Semantics-Based Approach for Cloned Vulnerable Code Detection. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Michalis Polychronakis and Michael Meier (Eds.). Springer International Publishing, Cham, 325–344.

[45] Yue Zou, Bihuan Ban, Yinxing Xue, and Yun Xu. 2021. CCGraph: a PDG-based code clone detector with approximate graph matching. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 931–942. doi:10.1145/3324884.3416541

Just Accepted