DEBUN: Detecting Bundled JavaScript Libraries on Web using Property-Order Graphs

Seojin Kim*
Sungkyunkwan University
001106ksj@gmail.com

Sungmin Park*
Korea University
ryan040@korea.ac.kr

Jihyeok Park†
Korea University
jihyeok_park@korea.ac.kr

***Abstract*—Detecting front-end JavaScript libraries in web applications is essential for website profiling, vulnerability detection, and dependency management. However, bundlers like Webpack transpile code in various ways, altering the original directory and code structure, which complicates library detection. While state-of-the-art techniques utilize property pattern-based library detection at runtime, they face two key limitations: (1) they cannot detect libraries inaccessible from the global object, and (2) they have limitations in granular version detection. To address these challenges, we present DEBUN, a scalable technique for detecting JavaScript libraries and their versions using function-level fingerprints. Our key insight is that bundlers preserve the property names and execution order of property operations, even after transpilation. To leverage this, we introduce the *property-order graph* (POG), which represents the execution order of property operations within a function body. We evaluate DEBUN on 68 high-traffic websites with 78 front-end JavaScript libraries. Our approach outperforms existing tools, achieving a 91.76% F1-score in library detection (1.39x higher) and an 79.81% F1-score in version identification with inclusion match (1.36x higher).**

I. 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 [1, 2]. Static analyzers also enhance precision and efficiency by incorporating predefined API modeling of detected libraries [3, 4]. According to W3Techs' survey, 98.8% of all websites use JavaScript, and 81.4% use third-party JavaScript libraries. Such an extensive use of libraries in web applications necessitates accurate detection techniques.

Challenges: However, detecting libraries in web applications is challenging because bundlers like Webpack or Rollup transpile the code and obscure its original structure. Developers use bundlers to minimize network requests when loading applications by reducing file size. 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 include at least one bundled code containing third-party libraries [5]. 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 [6, 7] 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 use common code clone detection techniques [8, 9, 10] used in other programming languages for 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) [11], employs this approach but with a manually defined set of detection patterns. Liu and Ziarek [12] introduce PTDETECTOR, a tool that automatically extracts *pTree* data structures, which represent the tree of reachable properties, to detect libraries. However, it has two key limitations: (1) it cannot detect libraries inaccessible from the global object, and (2) it lacks extensibility to version-level detection.

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' code with all possible configurations of bundlers and collect fingerprints to detect libraries [13]. It is inefficient because bundlers generate exponentially many different code depending on the configuration. If a bundler supports n boolean options, this approach generates at most 2^n fingerprints for each library.

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 observation 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 (§II), and propose how to construct consistent POGs with a *path-sensitive truthy analysis* (§III) based on abstract interpretation frameworks [14, 15]. Then, we detect libraries and versions by comparing POG-

* These authors contributed equally to this work.
† Corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00015

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

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

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

(b) Transpiled `arrayEach` function in TikTok website.

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

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. It outperforms existing tools, achieving a 91.76% and an 79.81% F1-score in library and version detection, respectively.

The contributions of this paper are as follows:

- We introduce a *property-order graph* (POG) to capture the execution order of property operations in function bodies, with consideration of control flow.
- We implement DEBUN to detect front-end JavaScript libraries in web applications by comparing POGs of target applications and libraries.
- We evaluate DEBUN on 68 high-traffic websites with 78 front-end JavaScript libraries and show that it outperforms state-of-the-art tools, achieving a 91.76% and an 79.81% F1-score in library and version detection.

II. MOTIVATION

This section shows the limitations of existing techniques in detecting libraries and their versions in web applications. It then presents the insight of our technique and the challenges with a real-world motivating example in Figure 1, showing the original `arrayEach` function in Lodash.js v4.17.21 and its transpiled code in the TikTok website.

A. Limitations of Existing Techniques

There are two main existing approaches to detect libraries or their versions in web applications: 1) property pattern-based detection at runtime and 2) exhaustive transpiled code search.

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) [11] detects Lodash.js[1] by checking the existence of the property `_` in the global object `win` and its child 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.

This limitation is addressed by the *pTree* data structure in the tool PTDETECTOR [12]. A *pTree* represents the tree structure of all reachable properties from the global object at runtime. While the tool automatically detects libraries by comparing them for the web application and libraries, it still has two key limitations. First, they cannot detect libraries internally imported by bundlers through inner modules or immediately-invoked function expressions (IIFEs). For example, the `arrayEach` function of Lodash.js is imported as an internal module in TikTok website:

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

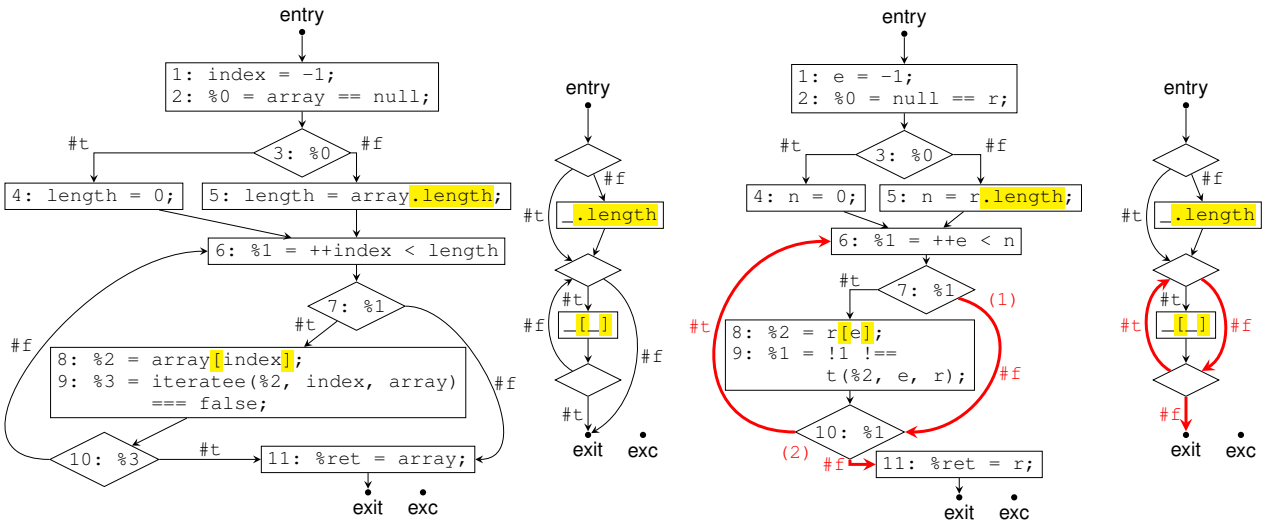
Second, property patterns are not sufficient to detect library versions when no version-specific property exists. Neglecting the code inside function bodies makes it difficult to distinguish different versions of the same library. A version update often changes the internal code structure of the library without changing the property pattern. 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]

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 a simple syntactic transformation [16, 17, 18] (e.g., formatting and variable renaming) or control/data flows [19, 20]. However, JavaScript significantly alter the code structure as well through advanced optimizations, which even change the control/data flow of the code. For example, Figure 1 shows the change of the code structure after transpilation, making it infeasible to naively apply existing code clone detection techniques.

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), the number of possible configurations grows exponentially (2^{34} for Terser), requiring large memory. For example, the tool URR [13] suffers from this inefficiency. Even though they use only 24,576 configurations of Webpack, their tool requires 2.27GB of memory only for three libraries. Thus, it is not scalable when detecting a large number of libraries in web applications.

[1] https://lodash.com/

[2] https://github.com/jquery/jquery/releases/tag/3.7.1



(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.

B. Our Approach: Property-Order Graphs

Instead, we focus on what bundlers retain in the code after transpilation. We first explain our key insight and then the challenges in applying it to real-world examples.

Key Insight: We observe that property names and execution order of property operations remain unchanged after bundling. It is reasonable because of the nature of JavaScript semantics regarding computed property names and the getter/setter for property operations. In JavaScript, properties can be accessed using computed names; `obj['h'+'i']` reads the property `hi` of the object `obj`. Thus, bundlers retain the property names considering the access of properties using computed names. 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 are often defined as getter and setters, respectively, with user-defined functions. It means that property read/write operations may implicitly call functions. Since the function 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];
}
```

because it has the exactly two same property read operations with the name `length` and the computed name `[_]`.

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*. We construct POGs by filtering only property operations with their names in the control-flow graph (CFG) of function bodies. For example, Figures 2a and 2c depicts CFGs of the original and bundled code in Figure 1. Each rectangle represents sequential normal instructions, and each diamond represents conditional instructions. 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: However, generated POGs in Figures 2b and 2d are still different (highlighted in thick red lines) because the CFG (Figure 2c) of the bundled code contains (1) an infeasible execution path and (2) a flipped conditional branch. First, a path $7 \rightarrow 10 \rightarrow 6$ is infeasible. The temporary variable `%1` is always falsy at `10` when the execution path comes from the false branch of `7`. It means that all the execution paths from the false branch of `7` are always flowed 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. It happens because transpilers often negate conditions and flip branches to minimize the code size. Such inconsistent CFGs cause different POGs as shown in Figures 2b and 2d and may cause false negatives in library detection.

Graphs	$\mathcal{G} ::= \{\overline{l : i}\}\ (l_{\text{entry}}, l_{\text{exit}}, l_{\text{exc}})$	
Instructions	$i ::= \texttt{if}\ (e)\ l\ \texttt{else}\ l$	
	$\mid\ x = e;\ l \mid x = e.p;\ l \mid e.p = e;\ l$	
	$\mid\ x = e[e];\ l \mid e[e] = e;\ l$	
Expressions	$e ::= c \mid x \mid\ !e \mid \{\} \mid e(\overline{e}) \mid e == e \mid e === e$	
	$\mid\ e < e \mid e > e \mid e \oplus e \mid \ominus e \mid \ldots$	
Constants	$c ::= b \mid n \mid z \mid s \mid \texttt{undefined} \mid \texttt{null}$	
Variables	$x \in \mathbb{X}$	**Properties** $p \in \mathbb{P}$
Labels	$l \in \mathbb{L}$	**Booleans** $b \in \mathbb{B} = \{\texttt{\#t}, \texttt{\#f}\}$
Numbers	$n \in \mathbb{N}$	**BigInts** $z \in \mathbb{Z}$
Strings	$s \in \mathbb{S}$	

Fig. 3: Control-flow graphs of JavaScript functions.

III. Construction of Property-Order Graphs

This section first introduces a basic construction algorithm of POGs (§III-A). To resolve the inconsistency in the basic algorithm, we introduce a *path-sensitive truthy analysis* (§III-B) and a three-step refinement of control flows in CFGs for consistent construction of POGs (§III-C).

A. Basic Construction Algorithm

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

1) CFG Construction: We use a standard algorithm to construct CFGs defined in Figure 3 where $\overline{A}$ 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 *conditional* instruction with two labels for branches.
- A *normal* instruction with its next label.

Each JavaScript loop statement (i.e., `while`/`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 §II-B, 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 omitted inequality/comparison operations are converted into equality/comparison operations 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`.

2) Property Operation Filtering: To construct POGs from CFGs, we filter 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 remain only whether they are read or write operations: $x = e_1[e_2]$ and $e_1[e_2] = e_3$ into $\_[\_]$ and $\_[\_] = \_$, respectively. When the computed properties names are constant strings, we treat them as property names: $x = e[\texttt{"name"}]$ into $\_.\texttt{name}$. For conditional instructions, we keep only their labels; $\texttt{if}\ (e)\ l_1\ \texttt{else}\ l_2$ into $\texttt{if}\ (\_)\ l_1\ \texttt{else}\ l_2$. Figures 2a and 2c depicts CFGs of the original and bundled code in Figure 1, and Figures 2b and 2d depicts POGs derived from the CFGs.

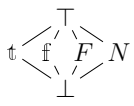
Path-Sensitive Results	$\widehat{\xi} \in \widehat{\Xi} = (\mathbb{L} \times \widehat{\Pi}) \to \widehat{\Sigma}$
Abstract Paths	$\widehat{\pi} \in \widehat{\Pi} = (\mathbb{L} \times \mathbb{B}) \uplus \{\bot\}$
Abstract States	$\widehat{\sigma} \in \widehat{\Sigma} = \mathbb{X} \to \widehat{\mathbb{V}}$
Abstract Values	$\widehat{v} \in \widehat{\mathbb{V}} = \{\bot, \mathbb{t}, \mathbb{f}, N, F, \top\}$

Fig. 4: Abstract domains for path-sensitive truthy analysis.

B. Path-Sensitive Truthy Analysis

We introduce a *path-sensitive truthy analysis* based on abstract interpretation [14, 15] to refine control-flows in CFGs for constructing consistent POGs. It analyzes the truthiness of each variable along each execution path partitioned by the latest conditional instruction.

1) Abstract Domains: Figure 4 shows the abstract domains for path-sensitive truthy analysis. A path-sensitive result $\widehat{\xi} \in \widehat{\Xi}$ is a mapping from pairs of labels and abstract paths to abstract states. An abstract path $\widehat{\pi} \in \widehat{\Pi}$ 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. An abstract state $\widehat{\sigma} \in \widehat{\Sigma}$ is a mapping from variables to abstract values.



An abstract value $\widehat{v} \in \widehat{\mathbb{V}}$ denotes 1) its truthiness ($\mathbb{t}$ or $\mathbb{f}$) or 2) whether it is flipped by negation operators (F or N). Its partial order ($\sqsubseteq$) and join ($\sqcup$) operations are defined with the lattice in the left Hasse diagram. 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 all other primitive values and objects are truthy values. The abstract values $\mathbb{t}$ and $\mathbb{f}$ means 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 conditional branches or side effects.

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

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

$$\textsc{Cond}\ \frac{\widehat{\xi}_1 = \mathsf{refine}(l,\widehat{\pi},\widehat{\sigma})(l_1, e, \#\mathtt{t}) \qquad \widehat{\xi}_2 = \mathsf{refine}(l,\widehat{\pi},\widehat{\sigma})(l_2, e, \#\mathtt{f})}{\widehat{[\![\mathtt{if}\ (e)\ l_1\ \mathtt{else}\ l_2]\!]}_i(l,\widehat{\pi},\widehat{\sigma}) = \widehat{\xi}_1 \sqcup \widehat{\xi}_2} \qquad \textsc{Assign}\ \frac{\widehat{[\![e]\!]}_e(\widehat{\sigma}) = (\widehat{v}, \widehat{\sigma}')}{\widehat{[\![x = e;\ l']\!]}_i(l,\widehat{\pi},\widehat{\sigma}) = \{(l',\widehat{\pi}) \mapsto \widehat{\sigma}'[x \mapsto \widehat{v}]\}}$$

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

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

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

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

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

$$\begin{array}{llll} \widehat{[\![e_1 == e_2]\!]}_e(\widehat{\sigma}) & = (N, \widehat{\sigma}) & \widehat{[\![\{\}]\!]}_e(\widehat{\sigma}) & = (\mathbb{t}, \widehat{\sigma}) \\ \widehat{[\![e_1 === e_2]\!]}_e(\widehat{\sigma}) & = (N, \widehat{\sigma}) & \widehat{[\![e(\overline{e_k})]\!]}_e(\widehat{\sigma}) & = (N, \widehat{\sigma}_\iota) \\ \widehat{[\![e_1 < e_2]\!]}_e(\widehat{\sigma}) & = (N, \widehat{\sigma}) & \widehat{[\![e_1 \oplus e_2]\!]}_e(\widehat{\sigma}) & = (N, \widehat{\sigma}) \\ \widehat{[\![e_1 > e_2]\!]}_e(\widehat{\sigma}) & = (N, \widehat{\sigma}) & \widehat{[\![\ominus e]\!]}_e(\widehat{\sigma}) & = (N, \widehat{\sigma}) \end{array}$$

Fig. 5: The abstract semantics for instructions $\widehat{[\![-]\!]}_i$ and expressions $\widehat{[\![-]\!]}_e$ in the path-sensitive truthy analysis.

2) Abstract Semantics: The initial path-sensitive result $\widehat{\xi}_\iota$ consists of a single mapping from the pair of the entry label l_{entry} and $\bot$ to the initial abstract state $\widehat{\sigma}_\iota$ initialized with all variables to N because their truthiness are not yet flipped.

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

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

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

where $\mathsf{inst}(l)$ is the instruction labeled by l in the CFG, and $\widehat{[\![-]\!]}_i$ is the abstract semantics for instructions defined in Figure 5. The helper function $\mathsf{refine} : (\mathbb{L} \times \widehat{\Pi} \times \widehat{\Sigma}) \to (\mathbb{L} \times \mathbb{X} \times \mathbb{B}) \to \widehat{\Xi}$ refines abstract paths and states where e is the condition and b denotes the true/false branch as follows:

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

It preserves the abstract path $\widehat{\pi}$ and the abstract state $\widehat{\sigma}$ when the condition e always holds (or always does not hold) under the abstract path $(l, \widehat{\pi})$ (the third case). If the condition e is a variable x (the fourth and fifth cases), it refines the abstract value of the variable x to truthy or falsy. Conditional instructions refine paths and states, while assignments propagate the abstract value; instructions with side-effects initialize all variables to N. A constant or a literal expression produces $\mathbb{t}$ or $\mathbb{f}$ according to the language semantics, a negation operator flips the truthiness, and other expressions produce N by default. If an expression has a potential side-effect (e.g., function call), it initializes the abstract state with N for all variables.

Example: To illustrate how the path-sensitive truthy analysis works, we provide a simple example in Figure 6 and explain the analysis steps.

- **(Label 1)** The analysis begins with an empty abstract state at the entry label, associated with the base path $\bot$.
- **(Label 2)** Evaluating `x = Math.random() < 0.5` yields an unknown truthy value; `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 $(3, \#\mathtt{t})$ with $\mathtt{x} = \mathbb{t}$.
- **(Label 5)** On the other hand, in the false branch, the path is $(3, \#\mathtt{f})$ with $\mathtt{x} = \mathbb{f}$.
- **(Label 6)** Although outside the `if` statement, the analysis preserves the refined abstract paths. Here, `z` is mapped to $\mathbb{f}$ under $(3, \#\mathtt{t})$ and to $\mathbb{t}$ under $(3, \#\mathtt{f})$, 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.

C. Analysis-Based Flow-Refinement

We refine CFGs in three steps using the result of path-sensitive truthy analysis to construct consistent POGs. Figure 7 shows the step-by-step refinement of the CFG of the bundled code in Figure 1b using the analysis result. In each step, the dotted edge denotes the removed original edge, and the thick red lines denote the modified/added part in each step.

```
/*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	$\bot$	$\mathtt{x} \mapsto N$

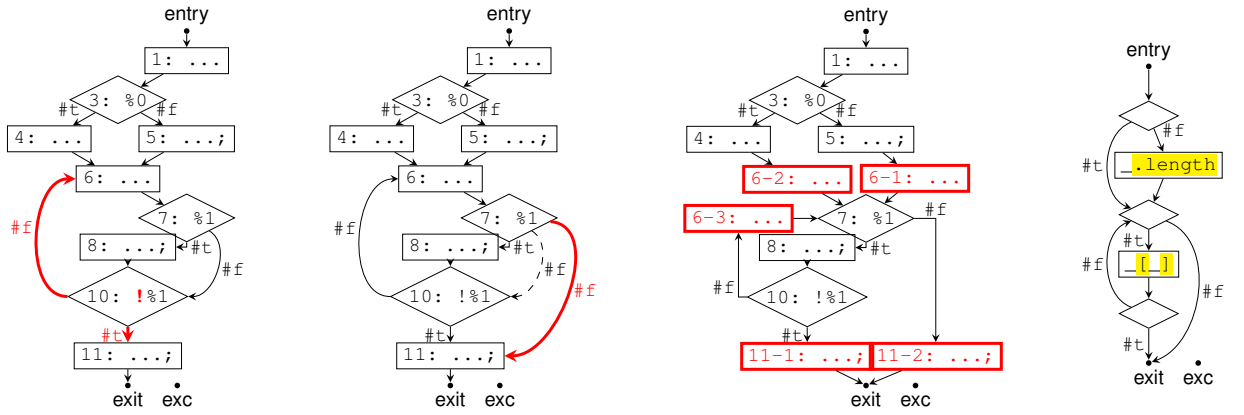
$\mathbb{L}$	$\widehat{\Pi}$	$\widehat{\Sigma}$
3	$\bot$	$\mathtt{x} \mapsto N$ $\mathtt{y} \mapsto F$
4	$(3, \#\mathtt{t})$	$\mathtt{x} \mapsto \mathbb{t}$ $\mathtt{y} \mapsto F$
5	$(3, \#\mathtt{f})$	$\mathtt{x} \mapsto \mathbb{f}$ $\mathtt{y} \mapsto F$

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

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

(b) The path-sensitive truthy analysis result.

Fig. 6: A simple example and its path-sensitive truthy analysis result.



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

Fig. 7: Step-by-step analysis-based flow refinement of the CFG of bundled code in Figure 2a and its POG.

Step 1: Branch Flipping: To reduce code size, transpilers often flip branches by negating the condition or its incoming expressions. For example:

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

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

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

For consistent branch flipping status, we flip a branch by negating the condition e of the branch if *more flipped* values are flowed into the branch than non-flipped ones. For example, to the condition `%1` at `10`, one flipped value is flowed from the true branch of `7` but no non-flipped value is flowed:

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

Thus, our approach flips the branch at `10` by negating the condition `%1` to `!%1` as shown in Figure 7a.

Step 2: Branch Bypassing: If a 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 branch at `7` (false) to branch at `10`, the condition `!%1` is always truthy:

$$(\mathtt{\%1} \mapsto \mathbb{f}) \in \widehat{\xi}(\mathtt{10}, (\mathtt{7}, \#\mathtt{f})) \quad \wedge \quad (\mathtt{\%1} \mapsto \mathbb{f}) \iff (\mathtt{!\%1} \mapsto \mathbb{t})$$

Thus, we can bypass the branch at `10` by connecting the false branch of `7` to `11` directly as shown in Figure 7b. It covers the common minification pattern using logical expressions:

`if(x){ if(y) return z; }` $\rightarrow$ `if(x&&y) return z;`

Step 3: Path Cloning: 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 true branch of `10` according to the analysis result:

$$\widehat{[\![\mathcal{G}]\!]}(\mathtt{6}, (\mathtt{3}, \#\mathtt{f})) \neq \bot$$

$$\widehat{[\![\mathcal{G}]\!]}(\mathtt{6}, (\mathtt{3}, \#\mathtt{t})) \neq \bot \qquad \widehat{[\![\mathcal{G}]\!]}(\mathtt{6}, (\mathtt{10}, \#\mathtt{f})) \neq \bot$$

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

$$\widehat{[\![\mathcal{G}]\!]}(\mathtt{11}, (\mathtt{7}, \#\mathtt{f})) \neq \bot \qquad \widehat{[\![\mathcal{G}]\!]}(\mathtt{11}, (\mathtt{10}, \#\mathtt{t})) \neq \bot$$

Thus, we clone the instruction at `6` and `11` for each abstract path as shown in Figure 7c. While it does not affect the POG in this example, path cloning is helpful to construct consistent POGs for the following minification patterns:

`if(y) x.p = e1; else x.p = e2;` $\rightarrow$ `x.p=y?e1:e2;`

After cloning the path, both have two property write operations for the property p in each branch, making the POGs consistent.

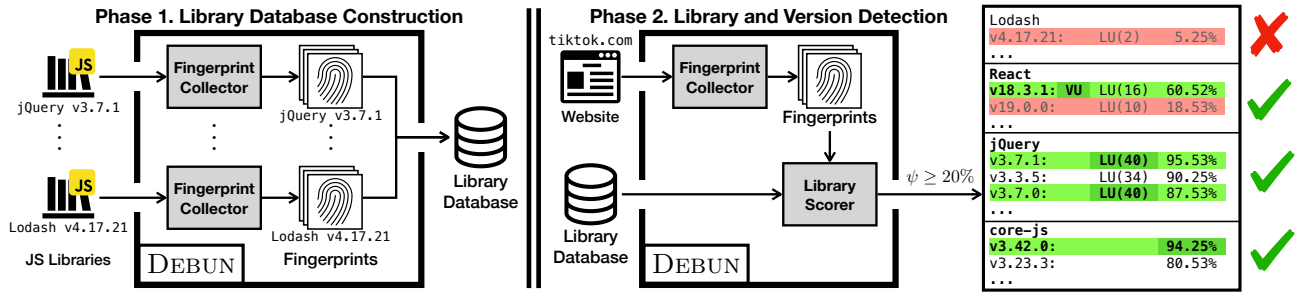


Fig. 8: Overall structure of DEBUN with two phases: 1) library database construction and 2) library (version) detection.

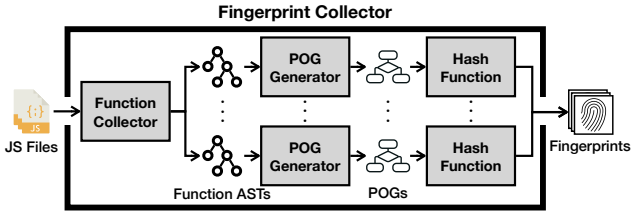


Fig. 9: The fingerprint collector for given JavaScript files.

IV. Overall Structure of DEBUN

Figure 8 depicts the overall structure of DEBUN with the core component called *fingerprint collector*.

A. Fingerprint Collector

The *fingerprint collector* extracts the fingerprints of all functions in given JavaScript files with three main components.

Function Collector: It extracts functions from JavaScript files by traversing the abstract syntax tree (AST) of the files using the `meriyah`[4] parser. To avoid redundancy, inner functions are stripped and split into separate functions.

```
function f(x) { var y = x.p; return () => y; }
```

For example, the above function splits into two functions: 1) `function f(x) { var y = x.p; return; }` and 2) `() => y`.

POG Generator: It generates a POG for each collected function using the process described in §III. We support general JavaScript syntax more than the syntax introduced in §III. First, we treat nullish coalescing operators (`??`) as conditional branches but do not apply branch bypassing and branch flipping. 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 of built-ins (e.g., `Math`, `Object`) because they have no side effects, allowing transpilers to safely transform them.

Hash Function: For each generated POG, we hash it using a fast hash function called rapidhash[5] to extract fingerprints of the functions. It computes the fingerprint of a POG node using its 1) node type, 2) property operations, and 3) child nodes. It starts with the root node and recursively hashes the child nodes, with memoization, to prevent redundant computations. To handle loops, visited child nodes are hashed with unique node id.

[4]https://www.npmjs.com/package/meriyah
[5]https://gitlab.com/fwojcik/smhasher3/-/tree/main/results

B. Phase 1: Library Database Construction

Using the fingerprint collector, we collect fingerprints of functions whose line of code is greater than five and construct a library database that maps fingerprints to library versions. We observe two sources of false positives: (1) overlapping functions between libraries [7, 21] arising when libraries partially include others (i.e., *shrinkwrapped clones*) or share common code patterns and (2) ubiquitous utility code injected by bundlers and shared across websites (e.g., bundler-generated loaders or polyfills). To mitigate (1), we apply *code segmentation* [21] with a *birth* (release date of a version). If more than 30% of functions overlap between two libraries, we retain the copy from the library version with the earliest birth date. To address (2) we remove from the database any function signature that appears on more than 60% of websites.

C. Phase 2: Library and Version Detection

Our tool computes the similarity score for each library version L_v by matching fingerprints with library database:

$$(\text{Score of } L_v) = \frac{(\#\text{ Matched Fingerprints of } L_v)}{(\#\text{ Fingerprints of } L_v)} \geq \psi$$

Then, we only retain library versions whose scores are greater than or equal to a given *score threshold* ψ. If at least one version of a library remains, we consider that the library exists. For each detected library, we decide its version based on *unique fingerprints* and scores. We compared two types of unique fingerprints: *library-unique fingerprints* (marked as `LU` in Figure 8), which are not shared with other libraries, and *version-unique fingerprints* (marked as `VU`), which are not shared with other libraries and versions. We apply the following rules, in order, moving to the next if the previous fails:

1) Select a version that contains version-unique fingerprints.
2) Select the version(s) with the highest number of library-unique fingerprints.
3) Select the version(s) with the highest similarity score.

V. Evaluation

We evaluate DEBUN with the following research questions:

- RQ1. **Library Detection:** Does DEBUN outperform LDC and PTDETECTOR in real-world websites?
- RQ2. **Library Version Detection:** Does DEBUN outperform LDC in real-world websites?
- RQ3. **Ablation Study:** Do POG-based fingerprints consistently and accurately represent functions?

All experiments were conducted on a server running an AMD Ryzen 9 7950X processor (16 cores, 32 threads, 4.5 GHz) with 128 GB DDR5 RAM. The system was equipped with an SK Hynix Platinum P41 2TB NVMe SSD and operated under a stable Linux distribution.

A. Data Collection

We collect JavaScript libraries from Cdnjs[6] and crawl bundled JavaScript files from high-traffic websites. Then, we manually construct a ground truth dataset indicating which libraries and versions are used in each website.

1) Library Collection: We collect 78 libraries used in PTDETECTOR with 8,256 versions, excluding libraries unavailable in Cdnjs and versions marked as pre-releases or nightly builds as they are rarely used in practice. When multiple libraries originate from the same GitHub repository, we treat them as a single library. For example, we treat `react` and `react-dom` as React.

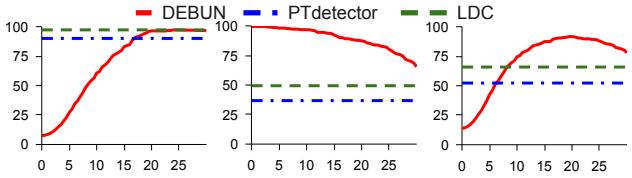
2) Website Collection: From the top 100 high-traffic websites listed by SEMRUSH,[7] we selected 68 sites, excluding those with crawling restrictions (e.g., reCAPTCHA). Then, we crawl JavaScript files from the websites with Puppeteer.[8] All data was collected on 27 February 2025.

3) Ground Truth Collection: Since no fixed ground truth dataset exists for library and version detection, we manually construct our own. We could not reuse PTDETECTOR's ground truth dataset because website updates alter the ground truth, and the websites' JavaScript codes used during their evaluation are no longer available. To build a reliable dataset, we combine the detection results from PTDETECTOR, LDC, and DEBUN with a conservative threshold ($\psi = 0.05$), then manually verify whether these libraries are actually used in the websites. We label a library if the code includes library-specific identifiers or at least five distinct functions, each longer than ten lines. While distinguishing between libraries was relatively easy, understanding and separating all 8,256 library versions was far more difficult. Even with careful manual inspection, partially imported libraries and mixed versions often make the exact version hard to determine. To ensure reliability, we assign version information only to libraries whose versions can be unambiguously verified through explicit indicators in the code (e.g., `_.version = "4.17.21"`), license texts, or comments. Based on this process, we construct a ground truth dataset[9] with 223 libraries, of which 105 have version annotations.

[6]https://cdnjs.com/
[7]https://www.semrush.com/website/top/
[8]https://pptr.dev/
[9]https://zenodo.org/record/15550954

TABLE I: Library detection scores when ψ = 20%.

Metric	LDC	PTDETECTOR	DEBUN
TP	111	82	195
FP	3	9	7
FN	112	141	28
Precision	97.37%	90.11%	96.53%
Recall	49.78%	36.77%	87.44%
F1-score	65.88%	52.23%	91.76%

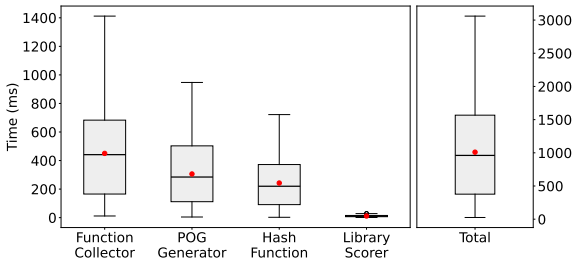


(a) Precision (%) (b) Recall (%) (c) F1-score (%)

Fig. 10: Library detection scores with different thresholds ψ.

B. RQ1. Library Detection

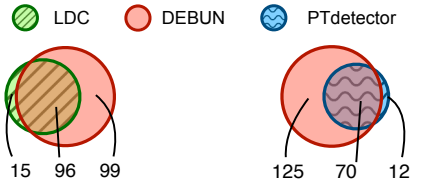
For comparison with library detection tools, we first collect the fingerprints using fingerprint collector (§IV). The library database occupies 31.97MB of memory and was constructed in 30 minutes. DEBUN detects libraries and their versions for each website in 1,009 ms on average, and the function collector consumes the most time due to the massive number of functions in the websites:



We compare the effectiveness of DEBUN with state-of-the-art library detection tools, LDC and PTDETECTOR. We set PTDETECTOR with its default setting, score threshold of 0.5 and depth limit of 3. We determine the optimal score threshold for DEBUN by varying it from 0% to 30% in 1% increments. Figure 10 shows the effectiveness by each ψ with true positive (TP), false positive (FP), and false negative (FN) counts. Note that we omit the true negative (TN) count and accuracy as they are far exceeded by TP, FP, and FN counts. The best score is obtained at ψ = 20% with the F1-score of 91.76%, which is 1.39x and 1.76x higher than LDC and PTDETECTOR, respectively. Two-tailed paired t-tests confirm these improvements are statistically significant (p = 1.86e-6 vs LDC, p = 6.10e-12 vs PTDETECTOR). Table I compares the scores of all tools at this threshold. DEBUN achieves higher precision than PTDETECTOR, though slightly lower than LDC, which is manually tuned. In terms of recall and F1-score, DEBUN outperforms all tools.

TABLE II: Comparison of the number of detected libraries.

Library	LDC	Δ	DEBUN	Δ	PTDETECTOR	Ground
React	13	+22	35	+34	1	35
core-js	33	-11	22	-8	30	35
Lodash.js	9	+17	26	+18	8	33
jQuery	27	+3	30	+8	22	30
Preact	3	+7	10	+10	0	10
Zepto	0	+10	10	+10	0	10
Total	**111**	**+84**	**195**	**+113**	**82**	**223**

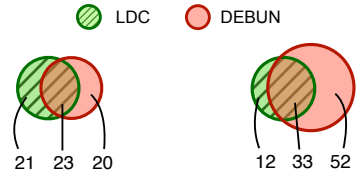


(a) DEBUN and LDC (b) DEBUN and PTDETECTOR

Fig. 11: Venn Diagram of the number of detected libraries.

TABLE III: Library version detection scores when $\psi = 20\%$.

	Exact		Inclusion	
Metric	LDC	DEBUN	LDC	DEBUN
TP	44	43	45	85
FP	0	10	3	23
FN	61	62	60	20
Precision	100.00%	81.13%	93.75%	78.70%
Recall	41.91%	40.95%	42.86%	80.95%
F1 score	59.07%	54.43%	58.82%	79.81%



(a) Exact match (b) Inclusion match

Fig. 12: Venn Diagram of the number of detected versions.

Recall: Table II shows the number of detected libraries by each tool, and Figure 11 shows its Venn diagram. DEBUN detects 84 and 113 more libraries than LDC and PTDETECTOR because they cannot detect libraries whose top-level properties are obfuscated (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. On the other hand, LDC and PTDETECTOR are good at detecting libraries partially imported into the global object (e.g., core-js and jQuery). For example, DEBUN fails to detect core-js on several websites where it is partially imported.

Precision: LDC demonstrates the highest precision due to the inclusion of various manual ad-hoc calculations. PTDETECTOR exhibits the lowest precision. PTDETECTOR struggles to distinguish libraries with overlapping property patterns. For example, Lodash.js and Underscore.js share a similar property pattern, _. Thus, the precision drops to 30% when evaluated only with Underscore.js. This suggests that property patterns are not distinguishable enough. While we mitigate the overlapping function issue by applying code segmentation (§IV-B), DEBUN still faces seven false positives. Of these, five stem from shared polyfill patterns and two from partial library imports (e.g., jquery-tools copies several functions from jQuery).

C. RQ2. Library Version Detection

We evaluate library version detection using two metrics: *exact match* and *inclusion match*. While different libraries have distinct fingerprints, versions of the same library often differ only slightly. Tree shaking may remove version-specific functions, making distinction harder. Moreover, 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. Thus, we consider a detection correct if it either exactly matches (exact match) or includes (inclusion match) the ground truth version. We compare the version detection effectiveness of DEBUN only with LDC because PTDETECTOR does not support version detection.

Table III shows the comparison of the version detection effectiveness of LDC and DEBUN. With the exact match, LDC and DEBUN correctly detect 44 and 43 versions, respectively. Statistical analysis shows no significant difference between the two approaches for exact matching (p = 0.96). LDC detects library versions only if the version label exists in the code. However, it struggles with libraries that do not have explicit version labels or have inconsistent version labels across versions. For example, 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. It results in many false negatives in LDC both in exact and inclusion match. On the other hand, DEBUN detects 85 versions (1.98x more than LDC) with the inclusion match, achieving a recall of 80.95% (1.89x higher than LDC). A two-tailed paired t-test confirms this improvement is statistically significant (p = 2.10e-16 $<$ 0.001). This demonstrates that POG provides high accuracy for version-unique or library-unique fingerprints in real-world, enabling accurate version identification without explicit version labels. However, challenges such as tree shaking and duplicated functions still hinder precise library version detection. Thus, leveraging both tools together, when possible, can lead to more accurate results.

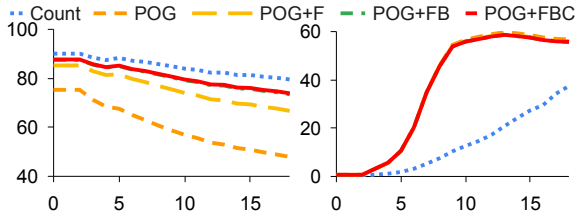
D. RQ3. Ablation Study

We evaluate the effectiveness of the POG-based function fingerprints by comparing different fingerprinting models:

- Count – Count per each property operation without order.
- POG – POGs via basic construction algorithm (§III-A).
- POG+F – POG with *branch flipping*.
- POG+FB – POG+F with *branch bypassing*.
- POG+FBC – POG+FB with *path cloning*.

TABLE IV: Scores for each model when LOC ≥ 6.

Metric	Count	POG	POG+F	POG+FB	POG+FBC
# Consistent	47,385	35,370	43,358	45,404	45,522
# Functions	54,368	54,368	54,368	54,368	54,368
Consistency	87.16%	65.06%	79.75%	83.51%	83.73%
# Functions	55,518	55,518	55,518	55,518	55,518
# Duplicated	1,715,034	274,252	273,252	273,678	273,684
Accuracy	3.28%	20.24%	20.32%	20.29%	20.29%



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

Fig. 13: Scores for each fingerprinting models with LOC $\geq x$.

For each fingerprinting model, we compute two metrics: *consistency*, the proportion of functions whose fingerprints remain unchanged both before and after transpilation, and *accuracy*, the ratio of distinct functions to the number of distinct fingerprints, indicating how well fingerprints differentiate between different functions.

$$\text{Consistency} = \frac{\text{\# Consistent}}{\text{\# Functions}} \quad \text{and} \quad \text{Accuracy} = \frac{\text{\# Functions}}{\text{\# Duplicated}}$$

We collect 256,884 function hashes from the latest versions of the target libraries. To reduce potential bias, we remove functions with identical syntax. This preprocessing yields a final dataset of 91,898 functions. We then transpile them using Terser and SWC with the most aggressive minify options except for unsafe options. Figure 13 compares model scores across different line-of-code (LOC) thresholds x. Table IV presents the details for LOC ≥ 6 as we collect fingerprints with LOC ≥ 6 to reduce noise from small functions (§IV-B).

Consistency: The Count model is most consistent, as it simply counts the number of property operations. In contrast, the baseline POG model exhibits the lowest consistency due to the inconsistency of the control flow explained in §II-B. The consistency continues to improve as control-flow refinement techniques are progressively applied, and the branch flipping has the greatest impact.

Accuracy: All POG-based models achieve significantly higher accuracy than Count, while showing similar effectiveness among themselves. This indicates that preserving the execution order of property operations, as in POG-based models, is effective in distinguishing functions after transpilation. In contrast, Count performs the worst, highlighting the importance of order information.

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 steps improve 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. To validate the effectiveness of our POG-based approach in real-world websites, we conduct an ablation study comparing three fingerprinting models: Count (simple property operation counts), POG (basic POG), and POG+FBC (POG with all refinements) on actual bundled JavaScript detection tasks.

TABLE V: Library detection scores across different models.

Metric	Count	POG	POG+FBC
TP	206	190	195
FP	66	8	7
FN	17	33	28
Precision	75.74%	95.96%	96.53%
Recall	92.38%	85.20%	87.44%
F1-score	83.23%	90.26%	91.76%

TABLE VI: Version detection scores across different models.

Metric	Count	POG	POG+FBC
TP	88	76	85
FP	79	32	23
FN	17	29	20
Precision	52.69%	70.37%	78.70%
Recall	83.81%	72.38%	80.95%
F1-score	64.71%	71.36%	79.81%

The results demonstrate how consistency and accuracy metrics directly impact detection performance. Table V shows that for library detection, Count achieves high recall but suffers from low precision due to poor accuracy, resulting in an F1-score of 83.23%. Conversely, POG shows high precision but lower recall due to consistency issues, yielding an F1-score of 90.26%. POG+FBC achieves the optimal balance with precision and recall, resulting in the highest F1-score of 91.76%. Table VI demonstrates that version detection shows even more pronounced differences in precision and recall, where POG+FBC outperforms both baselines with an F1-score of 79.81%, compared to 64.71% for Count and 71.36% for POG. These results confirm that our control-flow refinements effectively address the trade-off between consistency and accuracy, leading to superior detection performance in practice.

VI. Discussion

A. Advantages and Limitations

Advantages of DEBUN and POG: 1) **High recall**: While maintaining comparable precision, DEBUN achieves up to twice the recall of state-of-the-art tools. This indicates that DEBUN can detect twice as many libraries, demonstrating the effectiveness of POG as a fingerprint for bundled library detection. 2) **Fully automated library fingerprint collection**: Although PTDETECTOR is a state-of-the-art tool, it requires manual identification of library dependencies. It relies on dynamic execution to extract library fingerprints, which would fail if outer dependencies are missing. Furthermore, failure

to identify inner dependencies can result in false positives. In contrast, DEBUN is a fully automated static analysis-based approach, eliminating the need for manual efforts when adding new libraries or versions. 3) **Versatility**: POG-based function fingerprints are applicable not only to library detection, but also to any task requiring the identification of original code within bundled code (e.g., vulnerable function detection, code provenance analysis and license compliance checking).

Limitations of POG: In Section V, we observed that the consistency of POG is not 100%, despite our efforts to refine. This discrepancy can be attributed to *function inlining* during minification, where functions may be eliminated or their body moved inside other functions. While this transformation does not change the property access orders, our approach cannot handle inter-procedural execution paths, which can lead to false negatives. However, our evaluation shows that the impact of this limitation is not significant. In path-sensitive truthy analysis in Section III, the number of flipped and non-flipped values are used for branch flipping. In special cases where these numbers are equal, our approach may not effectively handle branch flipping. However, since transpilers gain no benefit from flipping when these numbers are equal, such behavior would be considered unusual.

B. Threats to Validity

1) Internal Validity: Our study has potential internal validity threats. First, the SEMRUSH top 100 may not represent all websites. Similarly, the libraries selected for evaluation may not fully represent all JavaScript libraries. While the moving target problem is another potential threat, we evaluated all tools at the same point in time to ensure a fair comparison. However, it was not feasible to reproduce the exact website states at the time of the prior work's evaluation. This mismatch may partly explain the performance drop of prior tools, both of which reported strong results in the past.

2) External Validity: We have observed that the same library can be used in multiple versions within a single website. For instance, on amazon.com, a comment indicated that jQuery v1.6.4 was being used, but certain functions had been upgraded to a newer version. Our current evaluation does not account for such version heterogeneity. Supporting this scenario would require additional mechanisms, such as fine-tuning the weights of version-unique or library-unique fingerprints.

VII. RELATED WORK

JavaScript Library Detection: Property pattern-based library detection techniques include LDC [11] and PTdetector [12]. While LDC requires a manual configuration of property patterns, PTdetector can automatically collect them. However, these approaches become entirely infeasible when runtime property patterns are modified during bundling process. Moreover, since these methods do not examine bundled code, they exhibit limitations in granular version detection. URR [13] detects vulnerable JavaScript libraries by exhaustive hash-based matching. To consider all of the possible transformations by bundlers, URR generates a large number of hashes.

Library Detection: Third-party library detection has been actively studied in other domains, such as iOS and Android apps. iOS approaches [22, 23] often rely on class-dump or binary analysis, which do not apply to JavaScript libraries. Detection techniques for Android third-party libraries [24, 25, 26, 27, 28] use structural features such as class or method signatures and opcode sequences, which make them resilient to code obfuscation. Approaches like ATVHunter [29] extract fingerprints from control flow graphs (CFGs) with execution order. However, detecting JavaScript libraries, especially in bundled code, requires refinement for control flow transformations introduced by JavaScript bundlers.

Code Clone Detection: Code clone detection techniques include text-based, token-based, tree-based, graph-based, and measure-based approaches. Text-based detection [16, 30], token-based detection [17, 31] and tree-based detection [18, 32] compare the similarity of source code text and syntax token. But they are not suitable for detecting bundled JavaScript libraries due to syntax transformations by JavaScript bundlers. Graph-based detection techniques [19, 20] compare the similarity of graphs like control flow graphs or program dependency graphs. These approaches assume that the control flow and data flow are preserved, which is not the case for bundled JavaScript libraries. Measure-based detection techniques [9, 10, 33, 34], measure the similarity of program features. These techniques often exhibit higher resilience to syntax changes but demonstrate lower precision, making them less suitable for JavaScript library detection in bundled contexts.

VIII. CONCLUSION

Detecting JavaScript libraries in modern web applications is challenging due to bundler transformations. Existing property pattern-based techniques fail to detect non-global libraries and distinguish versions. To address this, we proposed DEBUN, which utilizes function-level fingerprints from *property-order graphs* to identify libraries and their versions, even after code transpilation. Our evaluation on 68 high-traffic websites and 78 libraries demonstrates that DEBUN significantly outperforms existing techniques, achieving a 91.76% (1.39x higher) and an 79.81% (1.36x higher) F1-score in library and version identification, respectively.

DATA AVAILABILITY

The source code of DEBUN and the package for replicating the experimental results are available in the public repository: https://github.com/ku-plrg/debun-ase25.

ACKNOWLEDGEMENTS

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)

References

[1] C.-A. Staicu and M. Pradel, "Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 361–376. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/staicu

[2] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "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*, ser. NDSS 2017. Internet Society, 2017. [Online]. Available: http://dx.doi.org/10.14722/ndss.2017.23414

[3] S. Bae, H. Cho, I. Lim, and S. Ryu, "SAFEWAPI: Web API Misuse Detector for Web Applications," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 507–517.

[4] J. Park, "JavaScript API Misuse Detection by Using TypeScript," in *Proceedings of the Companion Publication of the 13th International Conference on Modularity*, ser. MODULARITY 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 11–12. [Online]. Available: https://doi.org/10.1145/2584469.2584472

[5] J. Rack and C.-A. Staicu, "Jack-in-the-box: An Empirical Study of JavaScript Bundling on the Web and its Security Implications," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 3198–3212. [Online]. Available: https://doi.org/10.1145/3576915.3623140

[6] A. Møller, B. B. Nielsen, and M. T. Torp, "Detecting locations in JavaScript programs affected by breaking library changes," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428255

[7] E. Wyss, L. De Carli, and D. Davidson, "What the fork? finding hidden code clones in npm," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 2415–2426. [Online]. Available: https://doi.org/10.1145/3510003.3510168

[8] R. Lin, Y. Fu, W. Yi, J. Yang, J. Cao, Z. Dong, F. Xie, and H. Li, "Vulnerabilities and Security Patches Detection in OSS: A Survey," *ACM Comput. Surv.*, vol. 57, no. 1, Oct. 2024. [Online]. Available: https://doi.org/10.1145/3694782

[9] S. Woo, H. Hong, E. Choi, and H. Lee, "MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3037–3053. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/woo

[10] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou, and W. Shi, "MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1165–1182. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/xiao

[11] GitHub, "Library detector for chrome (ldc)," 2025, accessed: 14-Mar-2025. [Online]. Available: https://github.com/johnmichel/Library-Detector-for-Chrome/

[12] X. Liu and L. Ziarek, "PTDETECTOR: An Automated JavaScript Front-end Library Detector," in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023, pp. 649–660.

[13] M. M. Ali, P. Snyder, C. Kanich, and H. Haddadi, "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*, ser. CCS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 2192–2206. [Online]. Available: https://doi.org/10.1145/3658644.3690262

[14] P. Cousot and R. Cousot, "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)*, 1977.

[15] ——, "Abstract Interpretation Frameworks," *Journal of Logic and Computation (JLC)*, vol. 2, no. 4, pp. 511–547, 1992.

[16] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 595–614.

[17] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "CCLearner: A Deep Learning-Based Clone Detection Approach," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 249–260.

[18] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A Novel Neural Source Code Representation Based on Abstract Syntax Tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 783–794.

[19] G. Zhao and J. Huang, "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*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 141–151. [Online]. Available: https://doi.org/

10.1145/3236024.3236068

[20] Y. Zou, B. Ban, Y. Xue, and Y. Xu, "CCGraph: a PDG-based code clone detector with approximate graph matching," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 931–942. [Online]. Available: https://doi.org/10.1145/3324884.3416541

[21] S. Woo, S. Park, S. Kim, H. Lee, and H. Oh, "CENTRIS: A precise and scalable approach for identifying modified open-source software reuse," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 860–872.

[22] Orikogbo, Damilola and Büchler, Matthias and Egele, Manuel, "Crios: Toward large-scale ios application analysis," in *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 33–42. [Online]. Available: https://doi.org/10.1145/2994459.2994473

[23] D. Domínguez-Álvarez, A. de la Cruz, A. Gorla, and J. Caballero, "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*, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1407–1418. [Online]. Available: https://doi.org/10.1145/3611643.3616344

[24] M. Backes, S. Bugiel, and E. Derr, "Reliable Third-Party Library Detection in Android and its Security Applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 356–367. [Online]. Available: https://doi.org/10.1145/2976749.2978333

[25] Y. Wang, H. Wu, H. Zhang, and A. Rountev, "ORLIS: obfuscation-resilient library detection for Android," in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 13–23. [Online]. Available: https://doi.org/10.1145/3197231.3197248

[26] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, and H. Chen, "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)*, 2018, pp. 141–152.

[27] J. Zhang, A. R. Beresford, and S. A. Kollmann, "LibID: reliable identification of obfuscated third-party Android libraries," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 55–65. [Online]. Available: https://doi.org/10.1145/3293882.3330563

[28] Y. Wu, C. Sun, D. Zeng, G. Tan, S. Ma, and P. Wang, "LibScan: Towards more precise Third-Party library identification for android applications," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 3385–3402. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/wu-yafei

[29] X. Zhan, L. Fan, S. Chen, F. We, T. Liu, X. Luo, and Y. Liu, "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)*, 2021, pp. 1695–1707.

[30] G. Mathew, C. Parnin, and K. T. Stolee, "SLACC: simion-based language agnostic code clones," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 210–221. [Online]. Available: https://doi.org/10.1145/3377811.3380407

[31] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1157–1168. [Online]. Available: https://doi.org/10.1145/2884781.2884877

[32] D. Zou, H. Qi, Z. Li, S. Wu, H. Jin, G. Sun, S. Wang, and Y. Zhong, "SCVD: A New Semantics-Based Approach for Cloned Vulnerable Code Detection," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, M. Polychronakis and M. Meier, Eds. Cham: Springer International Publishing, 2017, pp. 325–344.

[33] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "VulPecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ser. ACSAC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 201–213. [Online]. Available: https://doi.org/10.1145/2991079.2991102

[34] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "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*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 354–365. [Online]. Available: https://doi.org/10.1145/3236024.3236026