PTV: Better Version Detection on JavaScript Library Based on Unique Subtree Mining

XINYUE LIU and LUKASZ ZIAREK, University at Buffalo, USA

Identifying the versions of libraries used by a web page is an important task for sales intelligence, website profiling, and even security analysis. Recent work uses tree structure to represent the property relationships of the library at runtime, leading to more accurate library detection results. However, state-of-the-art tree-based detection methods cannot be directly used to detect specific versions of libraries. This paper proposes a novel algorithm to find the most unique structure out of each tree in a forest so that the size of the trees can be greatly minified. We show that an implementation of our algorithm in a state-of-the-art library detection tool, not only does not affect detection accuracy but reduces associated costs where tree-based detection methods can be used to detect library versions. Our experiment results show that our tool reduces space requirements by up to 99% and achieves better version detection for JavaScript libraries compared with existing tools.

CCS Concepts: • **Software and its engineering → Software libraries and repositories**.

Additional Key Words and Phrases: JavaScript Library, Dynamic Analysis, Version Detection, Tree Algorithm

1 INTRODUCTION

With the increase in the variety of sophisticated web applications, the demand for front-end libraries continues to grow. To illustrate this growth consider Cdnjs, the largest CDN (Content Delivery Network) that serves websites. Cdnjs now contains 6,056 different JavaScript libraries[1], almost twice as many as one year ago. With the staggering growth in JavaScript front-end libraries, there is an equal need for automatic library detection. JavaScript front-end library detectors are frequently used for competitor analysis, sales intelligence, security analysis, and website profiling.

Version detection is a crucial problem in library detection, especially for security analysis. Current static analysis methods for web applications require separate modeling of libraries[Sun and Ryu 2017]. Knowing the version of the library allows for more accurate modeling, thus leading to more reliable static analysis results. To illustrate the scope of the problem, a recent experiment on 5,000 of the top websites discovered that 76.6% of them include a vulnerability in a front-end library [Kadlec 2017]. The vast majority of these vulnerabilities were due to including an out-of-date version of a common library. Library version detection can efficiently and automatically identify frontend JavaScript applications that use library versions with potential risks.

Although JavaScript library detectors exist, unfortunately, there is no trivial way to determine the library version when a library is detected. Even though some libraries store their version as a string, allowing detectors to fetch and detect version based on this runtime value, this type of labeling is not comprehensive. In fact many libraries incorrectly label their own versions or do not consistently label their libraries. There is no standardized labeling format between different libraries. Current library detectors rely on manually collecting version label patterns for version detection. The most popular detector, LDC, can recognize versions for only 83 libraries. The most accurate detector, PTdetector [Liu and Ziarek 2023], uses tree structures to automate library feature extraction but cannot easily detect versions due to space requirements to store separate trees for each version.

In this paper, we build on the idea of using tree structures for library detection pioneered in PTdetector, and propose a new tool PTV (Shortened for "PTdetector-Version") to enable tree-based version detection for JavaScript Libraries. PTV is anonymously and publicly available here [GitHub

[1]Data source: https://cdnjs.com (Nov. 2023)

2023b] and we plan on packaging PTV in an artifact submission. Our paper makes the following contributions:

(1) a novel algorithm to minify trees used in library detection. Our idea is to extract the most unique structure out of each tree in the forest, reducing the content being saved and used for runtime detection. The correctness of the algorithm is proved and the time complexity is analyzed. This algorithm is not limited to the JavaScript library version detection problem and can be applied to any similar tree-based detection task.
(2) an implementation of our algorithm in PTV to minify every tree without affecting the detection ability of tree based library detectors.
(3) a comprehensive evaluation of the version detection ability of PTV against existing library detection methods. Our results show that PTV reduces the memory footprint by 99.33% without affecting the detection accuracy, and outperforms the existing methods on all libraries in our dataset.

2 BACKGROUND AND MOTIVATION

2.1 Front-end JavaScript Library

JavaScript libraries are commonly designed to adapt to different runtime environments. The APIs of the library are composed of functions wrapped in objects. These objects are registered in the global context of the browser runtime, allowing the APIs to be globally available. Listing. 1 shows simplified code from a popular library Lodash[2] as an example to present the details of this process.

```
1  (function() {
2      function lodash(value) {
3          return new LodashWrapper(value);
4      }
5      // Define properties
6      lodash.chain = function(value) {
7          var result = lodash(value);
8          result.__wrapped__ = value;
9          return result;
10     }
11     lodash.VERSION = '4.9.0';
12     ...
13     // Export lodash
14     window._ = lodash;
15 }.call(this));
```

Listing 1. Simplified Lodash Browser Initialization Steps.

Listing. 1 presents a few key steps of the Lodash initialization in the browser. Line 1 defines an anonymous function to wrap all the code, and line 2 defines the function `lodash(value)`, which will return an initialized object. Note that a function is also an object in JavaScript. Then in line 6 - line 12, various APIs (`chain`, `VERSION`, and others) are registered as `lodash` object properties. Finally, in line 14, the `lodash` object is exposed to the identifier _ in the global context, i.e., registered as a property of `window`[3].

[2]A modern JavaScript utility library delivering modularity and performance.
[3]Code running in a web page share single global object `window`.

2.2 Detection Methods

There are many web JavaScript library detectors on the market. Most of them act as browser extensions that detect loaded libraries by checking specific properties at runtime. In Sec. 2.2.1 we use the most popular open-source detector, Library-Detector-for-Chrome (LDC), to illustrate their detection mechanism on libraries and versions, as well as their drawbacks. In response to the problems of these traditional detectors, PTdetector is proposed in [Liu and Ziarek 2023]. This tool makes use of the runtime property tree structure to enable automated feature extraction and more accurate library detection, which is discussed in Sec. 2.2.2.

2.2.1 Library-Detector-for-Chrome (LDC). LDC is the most popular (based on GitHub star rating) open-source JavaScript library detector. It was created in January 2010 and is still being updated today. It has 600+ stars on GitHub [GitHub 2023a] and 10,000+ users on the Chrome Extension Store [Store 2023]. As a browser extension, LDC uses dynamic methods to detect libraries. Listing. 2 is a simplified JavaScript snippet of the LDC source code used to detect Lodash.

```
function testLodash () {
    var _ = window._,
        wrapper = _.chain(1);
    if ( _ && wrapper.__wrapped__) {
        return { version: _.VERSION || UNKNOWN };
    }
    return false;
}
```

Listing 2. LDC detects libraries by examining properties during the browser runtime.

Listing. 2 examines two JavaScript properties: `_` and `_.chain`, in the global context. If both of them exist, Lodash is assumed to be present, and the version is determined by the value in property `_.VERSION`. We call such property containing version information as the *version label*. Most detectors on the market today use similar detection methods. Such approach is straightforward and easy to implement, but has a number of drawbacks shown in prior work [Liu and Ziarek 2023]. First, confirming the existence of a library by a few properties may lead to false positives due to the commonness of global property conflicts in JavaScript. Second, such method can not handle libraries wrapped by web bundlers, such as Webpack.

Unfortunately, identifying library version by reading the version labels is not always reliable. The location and the pattern of the version label varies between libraries, even between versions in a single library. With the growing number of web libraries and versions, manually finding accurate version label patterns requires is infeasible. Over six thousands libraries are registered on Cdnjs, but LDC can support version detection for only 83 libraries. In addition, as we will show in our experiments (Sec. 5.3), not all JavaScript libraries are labeled with correct version information. Indeed, only half of the libraries in our dataset have comprehensive labeling for their versions, and half of libraries that contain version labels have incorrect labels!

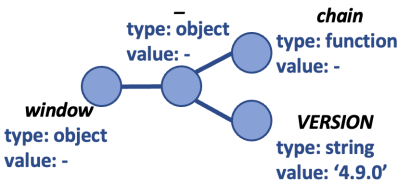

Fig. 1. pTree illustration of Lodash.

2.2.2 PTdetector. PTdetector introduces a new concept named *pTree*, which refers to a tree formed by the property relationship between JavaScript variables in a runtime frame. Each vertex in a pTree is assigned with the variable's name, type, and value. Every pTree is rooted at the global variable `window`. Fig. 1 shows a pTree generated from the Listing. 1.

PTDetector takes a JavaScript file and its dependency information as input and automatically extracts the runtime pTree as the detection feature using a trivial localhost client, and uses a weight-based tree-matching algorithm to score the existence of libraries on a web page. The rich details provided by the tree structure allow PTDetector to distinguish libraries more accurately and detect libraries wrapped by packers. This approach has a number of advantages over traditional methods, however, it does not support version detection.

2.3 Our Solution: pTree-based Version Detection

A naive, straightforward approach to enable pTree-based version detection is to generate a pTree for every version of every library. Following this idea, at browser runtime, we first use the pTree of the latest version of the library to determine if the library is loaded on the web page, as is done in PTDetector. After confirming the loaded library name and loaded location in the browser pTree, we then conduct tree matching against the pTrees of all versions of this library to determine which version has the best match. We discuss the challenges to this solution in the subsections below.

2.3.1 Correctness. Suppose that Lodash only has three versions – *A*, *B*, and *C*. Fig. 2 shows the pTrees for these versions. Consider if all vertices and edges of the pTree representing library version *A* are detected at runtime, can we conclude that the loaded version is *A*? Counter-intuitively, the answer is *no*. Consider that all vertices and edges in the pTree of version *A* also exist in the pTree of version *C*. Thus, we cannot tell if the loaded version of Lodash is *C* or *A*. We call the pTree of version *C* a *supertree* [4] of the pTree of version *A*. This situation is rather common in library version detection due to the high similarity in structures between library versions. One pTree may have multiple supertrees. In Sec. 3.5, we will introduce an algorithm identify and reason about supertrees.

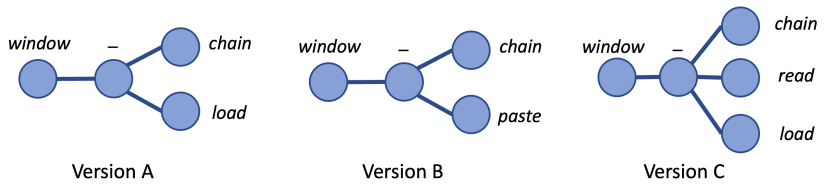

Fig. 2. Example of pTrees of different versions of Lodash. The type and value of each vertex are omitted in the figure. Assume that any vertices with the same name have the same type and value as well.

2.3.2 Memory Footprint. Today, there are 2,509,859 library versions on Cdnjs. According to the memory overhead estimation in the PTDetector paper, if we set the pTree node limit as 50, then over 8G space is needed to store all pTrees. Unfortunately, even a pTree with maximum 50 nodes is not enough to distinguish the subtle differences between the versions.

Our insight is to extract most unique structure out of each pTree, reducing the content being saved and used for runtime detection. For example, in Fig. 2, we can observe that the property `window._.chain` appears in all versions, so this property does not serve any distinguishing purpose in version detection, and should be discarded. In contrast, the property `window._.paste` only appears in version *B* and the property `window._.read` only appears in version *C*. Such property can completely substitute the functionality of the original pTree, being able to uniquely characterize the version. In other words, if the property `window._.paste` is detected during the runtime, we have confidence that the full pTree of version *B* can be detected. We call such structure as *unique*

[4]Similar to a *superset*. The formal definition of supertree will be given in Sec. 3.3

subtree. Following this intuition, we are able to design a method to minify every pTree without affecting the detection ability. In Sec. 3.4, we will present the tree minification algorithm that can find the unique subtree of each tree. Careful readers will notice that we are not able to find a unique subtree for the pTree of version A. This occurs because the pTree of version A has a supertree in our example. Before mining the unique subtree in a pTree, all its supertrees should be excluded. You will understand why it is safe to do so in Sec. 3.4.

3 ALGORITHM DESIGN

In this section we provide the core algorithms and their complexity analysis that underpin our implementation of pTree-based JavaScript library detection.

3.1 Basic Definition

3.1.1 Labeled Tree. We denote a labeled tree as $T = (V, E, \Sigma, L)$, consisting of a *vertex* set V, an *edge* set E, an *alphabet* Σ for vertex labels, and a *labeling function* $L : V \rightarrow \Sigma$. The *size* of T is the number of vertices in the tree.

A *path* is a sequence of vertices $p = (v_1, v_2, ..., v_n) \in V \times V \times ... \times V$ such that v_i is adjacent to v_{i+1} for $1 \leq i < n$. When the path's first vertex is root and the last vertex is a leaf, we call it a *full path*. For a tree T, we use $T.P$ to represent the set of all paths in T, and $T.P_f$ to represent the set of all full paths in T.

3.1.2 Induced Subtree. For a tree T with vertex set V and edge set E, we say that a tree T' with vertex set V' and edge set E' is an *induced subtree* of T, denoted as $T' \preceq T$, if and only if (1) $V' \subseteq V$, (2) $E' \subseteq E$, (3) The labeling of V' is preserved in T'. If $T' \preceq T$, we also say that T *contains* T'. Intuitively, an induced subtree T' can be obtained by repeatedly removing leaf vertices in T, or possibly the root vertex if it has only one child.

We say two trees T_1 and T_2 are *isomorphic* to each other, denoted as $T_1 = T_2$, if there is a one-to-one mapping from the vertices of T_1 to the vertices of T_2 that preserves vertex labels and adjacency. Based on the definition, we are easy to know that relation $\preceq$ is antisymmetric and transitive, i.e., $T_1 \preceq T_2$ and $T_2 \preceq T_1$ implies $T_1 = T_2$; $T_1 \preceq T_2$ and $T_2 \preceq T_3$ implies $T_1 \preceq T_3$. We use symbol $T_1 \prec T_2$ when $T_1 \preceq T_2$ but $T_1 \neq T_2$.

3.2 Problem Description

We can generalize the version detection problem in the following description. Assume there is a detection object labeled tree ϕ and a collection of detection samples, represented as a set of labeled trees $\Gamma = \{T_1, T_2, ..., T_n\}$.

In our practical problem, Γ is a collection of generated pTrees from one library under different versions, and ϕ is the detected library pTree generated during web page runtime. Each vertex in the pTree will carry extra information – name, value, and type - represented as labels mapping to vertices.

We say a tree in Γ is the *base tree* of ϕ if ϕ is grown from it though adding root and leaf vertices. For simplicity, we define the predicate $B_\phi(T)$: tree $T \in \Gamma$ is the base tree of ϕ. Based on our definition, we can deduce that the base tree has the following two properties.

(1) **Necessity:** if $B_\phi(T)$, then $T \preceq \phi$;
(2) **Uniqueness:** exact one $T \in \Gamma$ satisfies $B_\phi(T)$.

The first principle introduces the necessary condition of the base tree. If T is a base tree of ϕ, then T has to be an induced subtree of ϕ. The second principle claims that only one sample tree is the base tree. We want to find the exact base tree that the detection object tree ϕ is built from. And this matches our practical situation – a loaded library should only have one version.

Besides, the detection object tree ϕ is restricted by the following property:

Proposition 3.2.1. *Assume T_k is the base tree, we have $\forall p \in \phi.P$, if $p \notin T_k.P$, then $p \notin \bigcup_{T\in\Gamma} T.P_f$.*

This property indicates that during the ϕ growing process, i.e., when more vertices are added to the base tree to build ϕ, the newly created paths will not be the full paths already in sample trees in Γ. Intuitively, this property ensures that ϕ is not a mixture of multiple trees in Γ, otherwise there is no way to uniquely determine the base tree. This property holds in our detection task, because multiple versions of a library will be not loaded in the same place.

With these properties, the question is: given Γ and ϕ, how to find the $T \in \Gamma$, such that $B_\phi(T)$?

3.3 Simple Solution

First, in order to simplify later descriptions, here we make some additional definitions.

For two labeled trees T and T', if $T \preceq T'$, we say T' is a *supertree* of T; if $T \prec T'$, we say T' is a *strict supertree* of T. Given a tree set Γ, we use the symbol $\mathbb{S}(T)$ to represent the set of all supertrees of T with respect to Γ, named *supertree set*. In other words, $\mathbb{S}(T) = \{T' \in \Gamma \mid T \preceq T'\}$. Similarly, We use the symbol $\mathbb{S}_{st}(T)$ to represent the set of all strict supertrees of T.

We define the *equivalence class* of a tree T with respect to Γ as the set of all trees in Γ that is isomorphic to T, denoted as $[T]$, where $[T] = \{T' \in \Gamma | T' = T\}$. Easy to see that $[T] = \mathbb{S}(T) - \mathbb{S}_{st}(T)$. We provide an example for these definitions in Fig. 3.

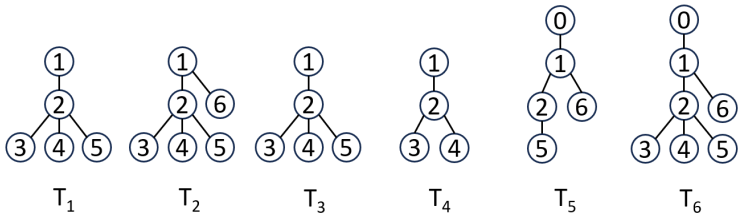

Fig. 3. Assume Γ consists of six trees in the plot, then we have $\mathbb{S}(T_1) = \{T_1, T_2, T_3, T_6\}$, $\mathbb{S}_{st}(T_1) = \{T_2, T_6\}$, $[T_1] = \{T_1, T_3\}$.

Furthermore, let's extend the predicate B_ϕ to B^*_ϕ when the variable is a set of trees. The $B^*_\phi(S)$ is defined as " $\exists\, T \in S$ such that $B_\phi(T)$"; and the $\neg B^*_\phi(S)$ is defined as " $\forall T \in S$, $\neg B_\phi(T)$". Based on this definition, we can reach two corollaries about B^*_ϕ.

Corollary 3.3.1. *For any two sets $S_1, S_2 \subseteq \Gamma$ that satisfy $S_1 \subseteq S_2$, if $B^*_\phi(S_2)$ and $\neg B^*_\phi(S_1)$, then $B^*_\phi(S_2 - S_1)$.*

Corollary 3.3.2. *For any two sets $S_1, S_2 \subseteq \Gamma$ that satisfy $S_1 \subseteq S_2$, if $\neg B^*_\phi(S_2)$, then $\neg B^*_\phi(S_1)$.*

Coro. 3.3.1 is due to the existence of the base tree – if the base tree does not exist in S_1, then it must be in $S_2 - S_1$. Coro. 3.3.2 describes that if the base tree does not exist in a set, then will not be in its subset as well. Both corollaries can be obtained directly from the definition of B^*_ϕ, so the proof is omitted. With these corollaries in hand, we can reach a vital proposition (Prop. 3.3.1), which enables us to determine which tree in Γ is the base tree through induced subtree judgment.

Lemma 3.3.1. *For an induced subtree T of ϕ, $B^*_\phi(\mathbb{S}(T))$.*

Proof. First let's prove that $\neg B^*_\phi(\Gamma - \mathbb{S}(T))$. Suppose $B^*_\phi(\Gamma - \mathbb{S}(T))$, which means that there exists a tree T' in Γ, such that $T \not\preceq T'$ and $B_\phi(T')$. From $T \not\preceq T'$, we know that there is a full path p of T not in the path set of T'. The lemma gives that T is an induced subtree of ϕ, so $p \in \phi.P$. Hence,

p is a *path* satisfies all the conditions in Prop. 3.2.1 but contradicts its conclusion – $p \notin \bigcup_{T\in\Gamma} T.P_f$. As a result, $\neg B^*_\phi(\Gamma - \mathbb{S}(T))$. Then with Coro. 3.3.1, because $B^*_\phi(\Gamma)$, we have $B^*_\phi(\mathbb{S}(T))$. □

Lemma 3.3.2. *For a tree T which is not an induced subtree of ϕ, $\neg B^*_\phi(\mathbb{S}(T))$.*

Proof. Suppose $B^*_\phi(\mathbb{S}(T))$, which means that there exists a tree T' in Γ, such that $T \preceq T'$ and $B_\phi(T')$. According to the necessity, $T' \preceq \phi$, so T is an induced subtree of ϕ (transitiveness). Contradiction. □

Proposition 3.3.1. *For an induced subtree T of ϕ, if $\forall T_s \in \mathbb{S}_{st}(T), T_s \npreceq \phi$, then $B^*_\phi([T])$; otherwise, $\neg B^*_\phi([T])$.*

Proof. If $\forall\, T_s \in \mathbb{S}_{st}(T), T_s \npreceq \phi$, then $\neg B^*(T_s)$ (Necessity). So $\neg B^*_\phi(\mathbb{S}_{st}(T))$. We know that $B^*_\phi(\mathbb{S}(T))$ because $T \preceq \phi$ (Lemma. 3.3.2). Then, based on Coro. 3.3.1, we have $B^*_\phi(\mathbb{S}(T) - \mathbb{S}_{st}(T)) \Rightarrow B^*_\phi([T])$.

Otherwise, if there exits a tree T_s in $\mathbb{S}_{st}(T)$, such that $T_s \preceq \phi$. Then based on Lemma. 3.3.1, we know $B^*_\phi(\mathbb{S}(T_s))$. So $\neg B^*_\phi(\Gamma - \mathbb{S}(T_s))$. Consider that $T \prec T_s$, so $[T] \subseteq \Gamma - \mathbb{S}(T_s)$, thus $\neg B^*_\phi([T])$. □

Prop. 3.3.1 shows that we can determine whether the base tree exists in T's equivalence class by checking $T \preceq \phi$ and all its strict supertrees' $T_s \preceq \phi$. In T's equivalence class, each tree is isomorphic to the other, so there is no method to tell who is the base tree. Ensuring the base tree is in a specific equivalence class is a satisfactory result for the problem. The algorithm to find the base tree in Γ is described as follows: iterate all trees in Γ, for each tree $T \in \Gamma$, check whether $T \preceq \phi$ and whether every strict supertree of it satisfies $T_s \preceq \phi$. Combined with Prop. 3.3.1, the result can be $B^*_\phi([T])$ or $\neg B^*_\phi([T])$. If the former is the case, then the algorithm terminates and the output is $[T]$). This algorithm ensures that the equivalence class in which the base tree is located will be found.

3.4 Unique Subtree Mining

Although we have given a deterministic algorithm to find the base tree, in our practical application scenarios, the sample trees (trees in Γ) are usually large and numerous. If the algorithm in the previous section is used for runtime detection, the time and space costs are unaffordable. As a result, in this section, we propose an algorithm to minify sample trees' size by unique subtree mining and ensure that the previous algorithm is still valid.

Algo. 1 shows the overall algorithm to reduce the size of sample trees. The input to the algorithm is the sample tree set Γ. For each $T \in \Gamma$, the output is its supertree set $\mathbb{S}(T)$ and a unique subtree T_m. We envision that T_m is a tree with a smaller size than T but the supertree set does not change. Namely, $\mathbb{S}(T_m) = \mathbb{S}(T)$. Note that the T in Prop. 3.3.1 is not required to be a member of the set Γ; this proposition applies whenever $T_m \preceq \phi$, so we can use T_m to replace the original T during the runtime detection. If $\mathbb{S}(T_m) = \mathbb{S}(T)$, then T_m must be an induced subtree of T, otherwise $T \in \mathbb{S}(T)$ while $T \notin \mathbb{S}(T_m)$. Therefore, our algorithm generates the specific T_m for each $T \in \Gamma$ by using a subset of the tree paths to reconstruct an induced subtree that satisfies $\mathbb{S}(T_m) = \mathbb{S}(T)$.

For each sample tree, Algo. 1 first calculates the path coloring, whose details are presented in Algo. 2. In Algo. 2 line 1, we initialize the coloring collection Ω as an empty set. For each full path in tree T, we use a coloring set ω to record the path's occurrence in other trees in Γ. If the full path occurs in the path set of another tree T_i, the tree's index i will be recorded in ω. Here we choose the full path instead of the normal path because we want to make sure the size of the generated T_m is large enough to prevent false positives during detection. In line 5, we combine all the coloring set ω of each full path together as a coloring collection Ω.

Algorithm 1 Unique Subtree Mining

Input: the sample tree set: Γ
Output: $\mathbb{S}(T)$ and the unique subtree T_m for each $T \in \Gamma$
1: **for** each $T \in \Gamma$ **do**
2: $\Omega \leftarrow PathColoring\,(T, \Gamma)$
3: $\mathbb{S}(T) \leftarrow \bigcap_{\omega \in \Omega} \omega$
4: Let $\overline{\Omega} := \{\Gamma - \omega \mid \omega \in \Omega\}$
5: $I \leftarrow MinCoverSet\,(\overline{\Omega}, \Gamma - \mathbb{S}(T))$
6: $T_m \leftarrow BuildTreeFromPath\,(T, I)$
7: **end for**

Algorithm 2 PathColoring

Input: the target set: Γ, a tree: $T \in \Gamma$
Output: a coloring collection Ω
1: Initialization: $\Omega \leftarrow \varnothing$
2: **for** each full path f in T **do**
3: let P_i represents the path set of each $T_i \in \Gamma$
4: f's coloring set $\omega := \{T_i \mid f \in P_i\}$
5: $\Omega \leftarrow \Omega \cup \{\omega\}$
6: **end for**

After getting the coloring collection Ω, in Algo. 1 line 3, the value of $\mathbb{S}(T)$ is obtained by intersecting all elements in the Ω.

Proposition 3.4.1. $\bigcap_{\omega \in \Omega} \omega = \mathbb{S}(T)$.

Proof. If a tree $T' \in \Gamma$ is in $\bigcap_{\omega \in \Omega} \omega$, then all the full paths of T is contained in the path set of T', so $T \preceq T'$; otherwise, at least one full path of T is not in the path set of T', so $T \npreceq T'$. As a result, $\bigcap_{\omega \in \Omega} \omega$ equals the set of all supertrees of T. □

The unique subtree T_m is generated from a subset of full paths of T. To ensure $\mathbb{S}(T_m) = \mathbb{S}(T)$, we need to find the smallest subset of Ω, such that the intersection of all its elements still equals $\mathbb{S}(T)$. If we complement both sides of the equation in Prop. 3.4.1, we can get $\bigcup_{\omega \in \Omega}(\Gamma - \omega) = \Gamma - \mathbb{S}(T)$ by De Morgan's laws. In this form, our question is equivalent to a well-known NP-complete problem – the *set cover problem* – which is described as follows.

> Given a set of elements $\{1, 2, \ldots, n\}$ (called the universe) and a collection S of m sets whose union equals the universe, the set cover problem is to identify the smallest sub-collection of S whose union equals the universe.

In our algorithm, the collection S in the description of the set cover problem is the inversed coloring collection $\overline{\Omega}$ defined in Algo. 1 line 4; and the universe U is $\Gamma - \mathbb{S}(T)$. In line 5, we invoke Algo. 3 to calculate the minimum cover subset of $\overline{\Omega}$. This algorithm will return an index set I, which contains the index of all elements that constitute the minimum cover subset. Using these indexes, in line 6, we construct the unique subtree T_m by invoking Algo. 4.

Algo. 3 is a famous greedy algorithm to solve the set cover problem within approximate polynomial time. At each stage, it chooses the set with the largest number of uncovered elements. This algorithm achieves an approximation ratio of $H(s)$, where s is the size of the set to be covered. In other words, it finds a set covering that may be $H(n)$ times as large as the minimum one, where $H(n)$ is the n-th harmonic number:

Algorithm 3 MinCoverSet

Input: a set collection: $S = \{\omega_1, \omega_2, ..., \omega_n\}$, the universe U
Output: a set $I \subseteq \{1, 2, ..., n\}$, such that $\bigcup_{i \in I} \omega_i = U$
1: Initialization: $I \leftarrow \varnothing$, $C \leftarrow \varnothing$
2: **while** $C \neq U$ **do**
3: Find the $i \in \{1, 2, ..., n\} - I$, such that $|C \cup \omega_i|$ is largest
4: $I \leftarrow I \cup \{i\}$
5: $C \leftarrow C \cup \omega_i$
6: **end while**

$$H(n) = \sum_{k=1}^{n} \frac{1}{k} \leq \ln n + 1 \tag{1}$$

Algorithm 4 BuildTreeFromPath

Input: a tree T with a path set $P = \{p_1, p_2, ..., p_k\}$, an index set $I \subseteq \{1, 2, ..., k\}$
Output: the unique subtree T_m
1: Initialization: $T_m \leftarrow \varnothing$
2: **for** each $i \in I$ **do**
3: Add path p_i to the tree T_m
4: **end for**

Algo. 4 shows the detail of T_m constructing. The input to the algorithm is a tree T and an index set I. We select the path whose index appears in the index set to construct the tree.

Lastly, let's use the trees in Fig. 3 to illustrate the whole unique subtree mining process. Here $\Gamma = \{T_1, T_2, T_3, T_4, T_5, T_6\}$. For T_1, firstly, we calculate the coloring of each full path of it in Algo. 2. There are three full paths and the corresponding coloring sets ω of T_1 shown in Table 1. Then we can get the value of $\mathbb{S}(T_1)$ by union all the ω. And the inversed coloring collection $\overline{\Omega} = \{\{T_5\}, \{T_5\}, \{T_4, T_5\}\}$. The Algo. 3 helps us to find the minimum sets from collection $\overline{\Omega}$ whose union equals $\Gamma - \mathbb{S}(T_1) = \{T_4, T_5\}$. And we can see that the combination of path $(1, 2, 3)$ and $(1, 2, 5)$ can meet the requirement. As a result, the unique subtree $(T_1)_m$ consists of these two paths. Similarly, we can get other unique subtrees. All unique subtrees are shown in Fig. 4.

Table 1. Unique subtree mining algorithm calculation result on T_1 in Fig. 3.

full paths	ω	$\mathbb{S}(T_1)$	$\overline{\Omega}$
$(1, 2, 3)$	$\{T_1, T_2, T_3, T_4, T_6\}$	$\{T_1, T_2, T_3, T_6\}$	$\{\{T_5\}, \{T_5\}, \{T_4\}\}$
$(1, 2, 4)$	$\{T_1, T_2, T_3, T_4, T_6\}$		
$(1, 2, 5)$	$\{T_1, T_2, T_3, T_5, T_6\}$		

3.5 Strict Supertree Set Minify

In Prop. 3.3.1 we need to verify all strict supertrees of T to determine whether the base tree is located in $[T]$; however, it is not necessary to iterate through the whole $\mathbb{S}_{st}(T)$. In Sec. 3.6, we will prove that the trees in the minified strict supertree set $\mathbb{S}_m(T)$, which is a subset of $\mathbb{S}_{st}(T)$, are all we need to check. Algo. 5 shows the process of generating $[T]$ and $\mathbb{S}_m(T)$ for each $T \in \Gamma$.

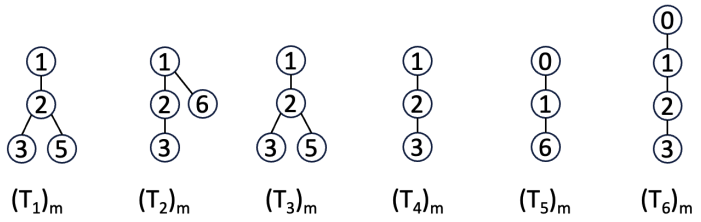

Fig. 4. The unique subtrees of trees in Fig. 3.

Algorithm 5 Strict Supertree Set Minify

Input: a tree T with its supertree set $\mathbb{S}(T)$
Output: $[T]$ and the minified strict supertree set $\mathbb{S}_m(T)$

1: Initialization: $[T] \leftarrow \emptyset$, $\mathbb{S}_m(T) \leftarrow \emptyset$
2: **for** each supertree $T' \in \mathbb{S}(T)$ **do**
3: **if** $T \in \mathbb{S}(T')$ **then**
4: $[T] \leftarrow [T] \cup \{T'\}$
5: **end if**
6: **end for**
7: $\mathbb{S}_{st}(T) := \mathbb{S}(T) - [T]$
8: **while** $\mathbb{S}_{st}(T) \neq \emptyset$ **do**
9: Find $K \in \mathbb{S}_{st}(T)$, such that $|\mathbb{S}(K)|$ is the largest
10: $\mathbb{S}_{st}(T) \leftarrow \mathbb{S}_{st}(T) - \mathbb{S}(K)$
11: $\mathbb{S}_m(T) \leftarrow \mathbb{S}_m(T) \cup \{K\}$
12: **end while**

In Algo. 5 line 1 - 6, we calculate $[T]$. The idea is simple: for a supertree of T, if $T \in \mathbb{S}(T')$ and $T' \in \mathbb{S}(T)$, then $T = T'$. Next, we get the value of $\mathbb{S}_{st}(T)$ in line 7 by removing isomorphic supertrees from $\mathbb{S}(T)$. From line 8, we start to generate the minified strict supertree set $\mathbb{S}_m(T)$. The algorithm always selects the element of $\mathbb{S}_{st}(T)$ that has the largest number of supertrees. This greedy algorithm ensures that $\mathbb{S}_m(T)$ holds the Prop. 3.5.1.

Lemma 3.5.1. *If $T \preceq T'$, then $\mathbb{S}(T') \subseteq \mathbb{S}(T)$.*

Proof. $\forall t \in \mathbb{S}(T')$, based on the definition of the supertree set, we know $T' \preceq t$. According to transitiveness, $T \preceq T' \preceq t$, so $t \in \mathbb{S}(T)$. Therefore, $\mathbb{S}(T') \subseteq \mathbb{S}(T)$. □

Proposition 3.5.1. *$\mathbb{S}_m(T)$ is the smallest subset of $\mathbb{S}_{st}(T)$ that satisfies $\bigcup_{K \in \mathbb{S}_m(T)} \mathbb{S}(K) = \mathbb{S}_{st}(T)$.*

Proof. We prove this using the greedy algorithm proof scheme.

(1) (Greedy Choice Property) Our greedy choice is K whose supertree set size is the largest in $\mathbb{S}_{st}(T)$. Suppose there is an optimal solution O that does not contain K. Because $K \in \mathbb{S}_{st}(T)$, there exists an K' in solution O such that $K \in \mathbb{S}(K')$; otherwise it can't satisfy $\bigcup_{K \in \mathbb{S}_m(T)} \mathbb{S}(K) = \mathbb{S}_{st}(T)$. So, $K' \preceq K$. Based on Lemma. 3.5.1, we know $\mathbb{S}(K) \subseteq \mathbb{S}(K')$. In our greedy choice, $|\mathbb{S}(K)|$ is the largest, so $\mathbb{S}(K) = \mathbb{S}(K')$. Hence, we can replace K' by K in O and still get an optimal solution.

(2) (Optimal Substructure Property) Let O be an optimal solution containing K. Consider the subproblem $\mathbb{S}'_{st}(T) = \mathbb{S}_{st}(T) - \mathbb{S}(K)$. We need to prove O contains the optimal solution for $\mathbb{S}'_{st}(T)$. Suppose $O - \{K\}$ is not an optimal solution for $\mathbb{S}'_{st}(T)$. We denote the optimal solution for $\mathbb{S}'_{st}(T)$ by O'. Then $|O'| < |O - \{K\}| = |O| - 1$. Given that $(\bigcup_{K \in O'} \mathbb{S}(K)) \cup \mathbb{S}(K) = \mathbb{S}_{st}(T)$, $O' \cup \{K\}$ is a solution with a smaller size than O. Hence, O is not an optimal solution. Contradiction. □

Take the trees in Fig. 3 as an example. The value of $\mathbb{S}_m(T_1)$ should be $\{T_2\}$, because $\mathbb{S}(T_2) = \{T_2, T_6\} = \mathbb{S}_{st}(T_1)$. Similarly, we have $\mathbb{S}_m(T_2) = \{T_6\}$, $\mathbb{S}_m(T_3) = \{T_2\}$, $\mathbb{S}_m(T_4) = \{T_1\}$, $\mathbb{S}_m(T_5) = \{T_6\}$, and $\mathbb{S}_m(T_6) = \varnothing$.

3.6 Runtime Detection

So far, for each $T \in \Gamma$, we get its unique subtree T_m in Sec. 3.4 and its minified strict supertree set $\mathbb{S}_m(T)$ in Sec. 3.5. Now, we can rewrite Prop. 3.3.1 in the following new version.

Proposition 3.6.1. *If $T_m \preceq \phi$ and $\forall\, K \in \mathbb{S}_m(T)$, $K_m \not\preceq \phi$, then $B^*_\phi([T])$; otherwise, $\neg B^*_\phi([T])$.*

Proof. We divide the condition into three cases.

(1) $T_m \not\preceq \phi$.

From Lemma. 3.3.2, we have $\neg B^*_\phi(\mathbb{S}(T_m))$. Due to $[T] \subseteq \mathbb{S}(T) = \mathbb{S}(T_m)$, we have $\neg B^*_\phi([T])$.

(2) $T_m \preceq \phi$, and $\exists\, K \in \mathbb{S}_m(T)$, such that $K_m \preceq \phi$.

From Lemma. 3.3.1, $B^*_\phi(\mathbb{S}(K_m))$. Because $\mathbb{S}(K_m) = \mathbb{S}(K)$, we have $B^*_\phi(\mathbb{S}(K))$, so $\neg B^*_\phi(\Gamma - \mathbb{S}(K))$. From the definition of $\mathbb{S}_m(T)$, we know $T \prec K$, so $[T] \subseteq \Gamma - \mathbb{S}(K)$. Therefore, $\neg B^*_\phi([T])$.

(3) $T_m \preceq \phi$, and $\forall K \in \mathbb{S}_m(T)$, $K_m \not\preceq \phi$.

Based on Lemma. 3.3.2, we can get $\forall K \in \mathbb{S}_m(T)$, $\neg B^*_\phi(\mathbb{S}(K_m))$, then $\neg B^*_\phi(\mathbb{S}(K))$. So we have $\neg B^*_\phi(\bigcup_{K \in \mathbb{S}_m(T)} \mathbb{S}(K))$, and this can be converted to $\neg B^*_\phi(\mathbb{S}_{st}(T))$ by Prop. 3.5.1. Furthermore, because $T_m \preceq \phi$, by Lemma. 3.3.1, we have $B^*_\phi(\mathbb{S}(T_m))$, thus $B^*_\phi(\mathbb{S}(T))$. Consequently, here comes $B^*_\phi(\mathbb{S}(T) - \mathbb{S}_{st}(T)) \Rightarrow B^*_\phi([T])$. □

Based on Prop. 3.6.1, given Γ and ϕ, we propose an algorithm to detect the base tree of ϕ in Γ, shown in Algo. 6. Let's say the original sample tree set is $\Gamma = \{T_1, T_2, ..., T_n\}$. Then the first input to the algorithm is a unique subtree set $\Gamma_m = \{(T_1)_m, (T_2)_m, ..., (T_n)_m\}$. We represent the indexes of the trees in Γ_m as $I = \{1, 2, ..., n\}$. Then, we define two mappings f_s and $f_e : I \to \mathcal{P}(I)$, where for an index $k \in I$, $f_s(k)$ maps to the set of all index of trees in $\mathbb{S}_m(T_k)$, and $f_e(k)$ maps to the set of all index of trees in $[T_k]$. Namely, $f_s(k) = \{i \mid T_i \in \mathbb{S}_m(T_k)\}$, and $f_e(k) = \{i \mid T_i \in [T_k]\}$. The last input to the algorithm is the detect object tree ϕ.

Algorithm 6 Runtime Detection

Input: the unique subtrees $\Gamma_m = \{(T_1)_m, (T_2)_m, ..., (T_n)_m\}$, two mappings f_s, f_e, and the detect object tree ϕ

Output: the indexes of possible base trees

```
1:  for each i ∈ {1, 2, ..., n} do
2:    if (T_i)_m ⪯ φ then
3:      for each j ∈ f_s(i) do
4:        if (T_j)_m ⪯ φ then
5:          go to 9
6:        end if
7:      end for
8:      return f_e(i)
9:    end if
10: end for
```

Algo. 6 guarantees to return the equivalence class of the base tree in Γ – it will traverse all equivalence classes to find the one that meets the condition in Prop. 3.6.1. Note that this algorithm

does not require the original sample trees Γ as input, resulting in faster speed and less space occupied during the detection runtime.

3.7 Algorithm Complexity

It is obvious that most part of the algorithm is in trivial linear time complexity. In this section, we only discuss two non-trivial parts – path coloring (Algo. 2) and minimum cover set (Algo. 3). Suppose there are n trees in Γ, and N vertices in Γ.

3.7.1 Path Coloring. To get path coloring, Algo. 2 iterates through all full paths in the tree and check whether these full paths appear in the path set of other trees in Γ. In our application, all the tree in Γ share a same root "window", and the "window" vertex will not appear at other places except root. Hence, given a full path f and a tree T, we only need at most $|f|$ times vertex comparisons to find out whether $f \in T.P$, where $|f|$ represents the number of vertices on the path f. Given a tree $T_1 \in \Gamma$, the time to calculate the path coloring of T_1 is:

$$n \cdot \sum_{f \in T_1.P_f} |f| \tag{2}$$

Observe that the number of full path in a tree is no more than its vertex number, and the vertex number of any full path is no more than tree's vertex number either. We have:

$$n \cdot \sum_{f \in T_1.P_f} |f| \leq n \cdot |T_1.P_f| \cdot |T_1.V| \leq n \cdot |T_1.V|^2 \tag{3}$$

Hence, the time to calculate the path coloring for all trees in Γ is:

$$\begin{aligned} T(\text{path coloring}) &= n \cdot \sum_{f \in T_1.F} |f| + n \cdot \sum_{f \in T_2.F} |f| + \cdots + n \cdot \sum_{f \in T_n.F} |f| \\ &\leq n \cdot \sum_{T \in \Gamma} |T.V|^2 \leq n \cdot (\sum_{T \in \Gamma} |T.V|)^2 = n \cdot N^2 \end{aligned} \tag{4}$$

So the time complexity of path coloring algorithm is $O(n \cdot N^2)$.

3.7.2 Minimum Cover Set. In Algo. 2, the size of set collection S equals the number of full path number. In each iteration, algorithm traverse all elements in S, and there are at most $|S|$ iterations. So, for a tree T, it requires at most $|S|^2$ operations to find the minimum cover set. The time to calculate the minimum cover set for all trees in Γ is:

$$T(\text{Minimum Cover Set}) = \sum_{T \in \Gamma} |T.P_f|^2 \leq \sum_{T \in \Gamma} |T.V|^2 \leq (\sum_{T \in \Gamma} |T.V|)^2 = N^2 \tag{5}$$

So the time complexity of minimum cover set algorithm is $O(N^2)$.

In conclusion, the overall tree processing algorithm has $O(n \cdot N^2)$ worst-case time complexity, where n is the number of tree, and N is the total number of vertex of all trees.

4 IMPLEMENTATION

We implement our algorithm into a Chrome extension named PTV. Fig. 5 shows the workflow of PTV library feature generation. For a library, first, we use PTDetector to generate the pTree for each of its versions. Inner dependency and outer dependency of each version are required as input to PTDetector to eliminate the dependency impact[5]. Outer dependency can be easily fetched on libraries' official sites, while inner dependency can only be inferred by reading library raw code,

[5]More discussion about inner dependency and outer dependency refers to [Liu and Ziarek 2023] Sec.III.C.(1).

which is time-consuming. However, for the version detection usage, inner dependencies will not only have no impact on the accuracy of the detection, but will also provide more information to allow us to differentiate versions. So, for each version of a library, we only provide its outer dependency information. In addition, we made some modifications to the pTree generation process. In the pTree generated by PTDetector, the vertex of the "array/set/map" type is stored with the number of elements as the value to avoid large trees. Since this strategy does not consider the actual elements of the data structure it represents it does not provide effective differentiation of such type of vertices. To account for the values stored in such data structures in the pTree, we modify PTDetector to use the MD5 checksum value of JSON stringified "array/set/map" variable as the vertex value.

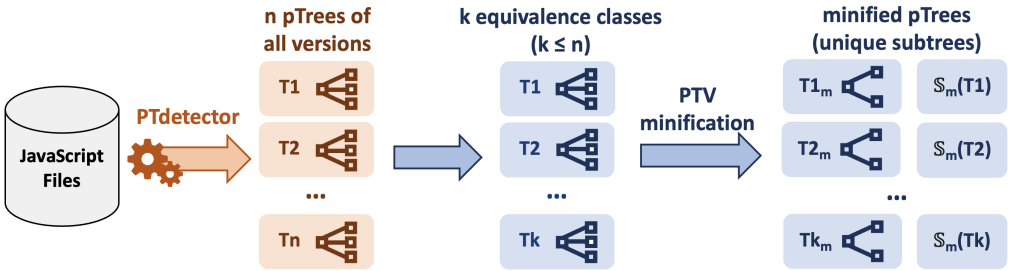

Fig. 5. PTV library version feature generation workflow.

Once we have the pTree for each version of each library we calculate the equivalence class of pTrees. When an isomorphisms exist between pTrees in large numbers, prioritizing the computation of equivalence classes can effectively decrease the values of n and N in the $O(n \cdot N^2)$ complexity of the path coloring algorithm (Algo. 2). Prior work [Jayapaul et al. 2015] shows that at most $n^2/m + n$ equality comparisons are sufficient to find all equivalence classes for n elements, where m is the largest size among all equivalence classes. Using their algorithm, we can shrink n pTrees to k unique pTrees ($k \leq n$). Then we apply our algorithms in Sec. 3.4 and Sec. 3.5 to generate k minified pTrees (unique subtrees) and their minimum supertree sets. We then save them in a local file for detection. The original pTree of the library's latest version will be also stored.

PTDetector is implemented as a Chrome extension that identifies libraries in the browser at runtime. We modify its detection process to enable version detection as given in Fig. 6. For a target web page, PTDetector will make use of libraries' latest version pTrees to identify loaded libraries and their root locations $\mathcal{X}$ in the browser runtime pTree. Then we apply Algo. 6 using minified pTrees and their minimum supertree sets information to identify the specific library version. Here the detect object tree ϕ in Algo. 6 is the pTree rooted at $\mathcal{X}$.

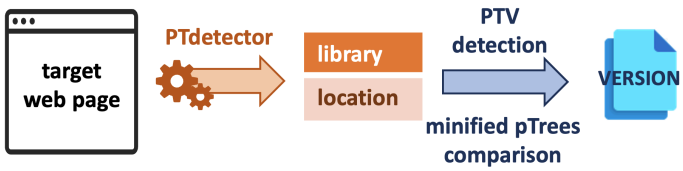

Fig. 6. PTV library version detection workflow.

5 EVALUATION

In this section, we evaluate the detection ability of PTV by answering four research questions.

5.1 Experiment Setup

In our experiment, we choose the most widely-used open-source web detector LDC as our baseline. To date, LDC can detect 125 libraries, and 83 of them have version detection information. Among 125 libraries, we selected 75 libraries with files mounted on Cdnjs, and excluded three frameworks – React, Vue, and Preact. As explained in the PTDetector paper, the code of these frameworks mounted on CDN is commonly their runtime debugging tool, which is optional to load. As a result, PTDetector has unsatisfactory detection on these frameworks, and it's meaningless to consider them in our experiment. For the libraries that are left, we manually investigate the outer dependencies for each version. Some libraries fail to load due to not being designed to run on the web client side, and some due to missing unknown dependencies. This leaves us with a set of 64 libraries that successfully run on our trivial local website, containing 3,664 versions in total. This set of libraries and all of their versions will be the dataset for our experiments.

All the experiments are conducted on macOS Sonoma (V 14.1.1) with an Apple M1 chip and 8G memory. All the web pages are opened on Chrome 118.0.5993.88 (Official Build) (arm64). This configuration is close to how everyday users use the browser and will represent realistic numbers of our tool "in the wild".

5.2 RQ1: How well is the minification of PTV?

We take these 64 libraries as our experiment target and generate a pTree for each version of them. For each pTree we set the depth limit as four and the size [6] limit as 1000 [7]. Our result shows that the average size of generated pTrees is 354 so a limit of 1000 is reasonable. After applying PTV, we generate minified pTrees for every library, with the average size of pTree as 3.4. The total size of detection required pTrees for 64 libraries is reduced from 1,256,094 to 8,404. Our algorithm reduces the memory footprint by 99.33% without affecting the detection accuracy (as will be shown in subsequent RQs).

On average, 8B space is required to store one pTree vertex in zipped JSON format. After minification, if one needs to realize the version detection on all libraries recorded on Cdnjs, it expects at most $2,509,859 \times 3.4 \times 8B = 65.45MB$ space to store all version classification features. The space to store the $\mathbb{S}_m$ of each minified pTree is negligible – less than 1% pTrees have nonempty $\mathbb{S}_m$ according to our experiment.

RQ1 Conclusion: PTV greatly reduces the size of the version pTree (99.33%), thus making pTree-based version detection possible. 65 MB is sufficient for all libraries on Cdnjs.

5.3 RQ2: Is the result of PTV sound?

5.3.1 Minified pTree Pattern. After manually analyzing minified pTrees produced from our dataset, we observe that it is a common practice for web library developers to store the version information in a specific property. For example, all jQuery versions store the version as a string value in the "`window.jq.fn.jquery`" property. So library detectors could determine the existence and version of jQuery by checking the value of this property during web runtime. We call such property containing version-related information as the *version label*, and the library version with version label as *explicit-labeled*.

For the explicit-labeled library version, the generated minified pTree will normally be the exact path of this property containing version information, because the version value is unique among

[6] *Size* refers to the number of vertices in a tree.

[7] It is not hard to infer that when every pTree of one library is trimmed based on the same depth limit, all the properties of the minification still hold. However, this is not true for the size limit trimming. In practice, we need a size limit to avoid extreme cases.

versions. Sometimes such a property will be given a different name, such as "`release`" or "`build`", and their locations vary from library to library. But it is easy to find such version storage patterns for each library by using our tool. Among 64 libraries, we found 33 of them have all versions explicit-labeled, 14 of them have some versions explicit-labeled, and others have no labeled version. We observed that many libraries don't contain version information in the initial versions, but add one when more versions are developed. Table 2 shows the list of libraries under these three categories. The value in the parentheses for libraries belonging to the second category is the percentage of explicit-labeled versions.

Table 2. Categorization of all libraries in our dataset based on explicit-labeled feature.

Category	Number	Libraries
Fully explicit-labeled	33	Backbone, D3, Dojo, Ext JS, Highcharts, InfoVis, jQuery, Qooxdoo, jQuery Tools, jQuery UI, Mapbox, Moment.js, MooTools, Prototype, RequireJS, Sammy, Underscore, Leaflet, three.js, Lo-Dash, Numeral.js, DC.js, Paper.js, Mustache, Moment Timezone, Pixi.js, Sea.js, Two, knockout, Fabric.js, Handlebars, Modernizr
Partly explicit-labeled	14	Raphaël (97%), YUI 3 (91.3%), FlotCharts (89%), Handsontable (80%), CamanJS (72%), Riot (70%), Matter.js (68%), core-js (67%), Processing.js (61%), jQuery Mobile (60%), FuseJS (54%), Socket.IO (25%), Pusher (25%), Velocity (10%)
None explicit-labeled	17	AmplifyJS, FlexSlider, FuseJS, SPF, IfVisible.js, Material Design Lite, FastClick, Isotope, yepnope, Tween.js, LABjs, Closure Library, Zepto.js, Head JS, Matter.js, Visibility.js

There are also cases where one library has more than one version storage pattern. Core-js is such an example. Core-js is the most popular library that provides polyfills for JavaScript, which allows modern code to run in old browsers. Core-js has over 200 versions. Before 1.2.0, there is no property storing the version information. Since 1.2.0, the version string can be found in "`window.core.version`". But from 3.10.0, "`window.core`" is replaced by "`window.__core-js_shared__`", and all version information is stored in a list instead of in a string variable.

5.3.2 Soundness. Confirming whether the detectors are capable of giving correct detection results is crucial, both for identifying the library as well as the specific version of the library. To test this, we set up an empty local web page to load each version of each library in the dataset sequentially and record the detection results of PTV and LDC on the web page. The detection result is normally a range instead of a single version. We use $\mathbb{V}_x$ to represent the set of all versions for a given library x. Then we can use the function $d_x : \mathbb{V}_x \to \mathcal{P}(\mathbb{V}_x)$ to denote the detection mapping relationship of the detector on library x. The domain of d_x is all the versions of library x we provided in the testing web server, and $d_x(v)$ is a set of versions, denoting the detection result on version $v \in \mathbb{V}_x$ of library x.

Using the detection results in our experiment, we can build the detection mapping d for both detectors. We specify for any $v \in \mathbb{V}_x$, when the detection result is "unknown" for the version but the library is correctly identified, in this case $d(v)$ should be $\mathbb{V}_x$, i.e., all versions may be true. When the library fails to be detected, $d(v)$ should be $\emptyset$. To illustrate, suppose there are five versions of core-js in our experiment dataset – "2.7.0", "2.8.0", "2.9.0", "3.0.0", and "3.1.0". Table 3 presents the value of $d_{core-js}(v)$ under different results.

We say a detection d_x is a *sound* mapping on version v if $v \in d_x(v)$. Based on our definition, detection $d_{core-js}$ is sound on versions "2.8.0", "2.9.0", and "3.0.0". According to the algorithm in Sec. 3, for any version v, PTV's detection $d(v) = [v]$. Since, $v \in [v]$, the detection result provided by

Table 3. An example to show how to build $d(v)$ based on the detection results.

core-js version v	detection result of LDC	$d_{core-js}(v)$ of LDC
2.7.0	library not detected	$\varnothing$
2.8.0	unknown version	{2.7.0, 2.8.0, 2.9.0, 3.0.0, 3.1.0}
2.9.0	2.9.0	{2.9.0}
3.0.0	$\geq$ 3.0.0	{3.0.0, 3.1.0}
3.1.0	$<$ 3.0.0	{2.7.0, 2.8.0, 2.9.0}

PTV is guaranteed to be sound. This conclusion is also well supported by our experimental results shown in subsequent RQs.

However, the situation is quite different when comes to LDC. Unexpectedly, we find that among 47 libraries with version information, 23 of them contain unsound mappings by LDC. We speculate one common reason is that the library developers forget to update the version property in a newer version. For instance, the value stored in the "`window.core.version`" of the core-js v1.2.7 is "1.2.6", a version ahead of the correct one. It is not uncommon that several continuous versions share the same version label. We call the explicit-labeled version that is assigned with a false version label the *mislabeled* version. Among the total 2,710 explicit-labeled library versions, we find 151 (5.6%) of them are mislabeled.

The mislabeling can lead to unsound mappings for detectors that naively read version information directly from these properties. We counted the unsound mappings by LDC for each library and show them in Fig. 7. Among 23 libraries containing unsound mappings, most libraries have less than ten unsound mappings, while library "YUI 3" and "FlotCharts" have rather high unsound mappings – 37 and 40 respectively. Upon manual inspection, the version management of both libraries is quite chaotic – more than half of the versions are stored with incorrect version information. Unsound detections in LDC vary from small libraries with less than 4k Github stars – "Raphaël", "Moment Timezone", "Processing.js", to popular and well-maintained libraries with more than 40k Github stars – "Lo-Dash", "Socket.IO", "Pixi.js". One conclusion is that incorrectly labeled version information is common among web libraries, and determining the version by reading the version property is not as reliable as we may think.

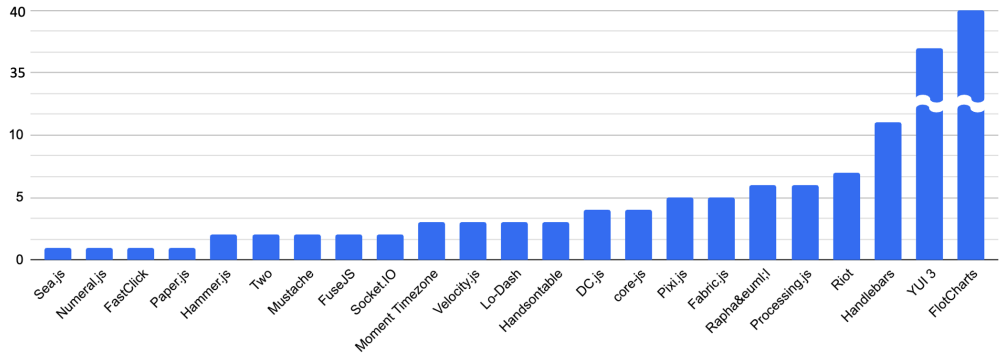

Fig. 7. The unsound detection mapping number of 23 libraries by LDC.

RQ2 Conclusion: PTV is guaranteed to be sound, while LDC is not. Among 47 libraries with version labels, 23 of them have unsound detection results using LDC due to version mislabeling by developers.

5.4 RQ3: How accurate is the classification ability of PTV?

To answer this question, we propose a new concept called *fineness* to quantify the *completeness* of detectors. Fineness is defined by the following equation.

$$fineness = \frac{|\{d_x(u) \mid u \in d_x(u)\}| - |\{d_x(v) \mid v \notin d_x(v)\}|}{|\mathbb{V}_x|} \tag{6}$$

Fineness is designed to characterize the classification ability of the detector on one library under the sound requirement – for all versions in a library, the more classes the detector soundly divides, then the larger the fineness. Intuitively, fineness is a measure of precision both in terms of accurately detection the library, but also correctly identifying the version. The numerator of Eq.(6) calculates the difference value between sizes of the sound detection and unsound detection. And the denominator is the number of versions of library x. The value of fineness ranges from -1 to 1. In the best case, $\forall\, v \in \mathbb{V}_x, d_x(v) = \{v\}$, then $fineness = 1$. When all versions are classified into the "unknown" class, $fineness = 1/|\mathbb{V}_x|$, i.e., correct but imprecise. In the worst case, unsound classification number is larger than the sound number, then $fineness < 0$. For a given library, we use the mark $F(PTV)$ to represent the fineness of PTV on this library, and the mark $F(LDC)$ to represent the similar of LDC.

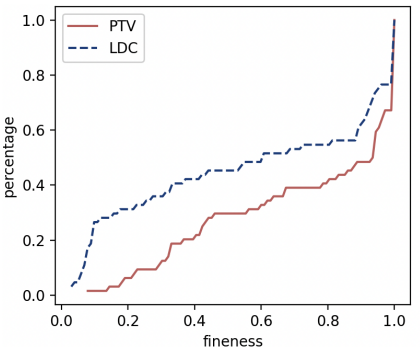

Fig. 8. CDF comparison of two detectors on 64 libraries.

Fig. 8 shows the cumulative distribution function (CDF) plot of $F(PTV)$ and $F(LDC)$ on our full dataset. The CDF illustrates what percent of the total number of libraries whose fineness is equal to or less than a given value. Both $F(PTV)$ and $F(LDC)$ are positive on all libraries. In general, PTV has a higher fineness in the higher value range compared with LDC.

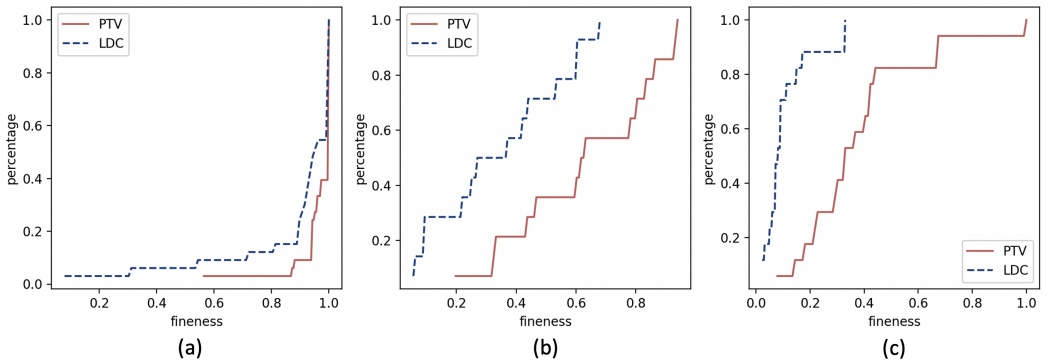

Fig. 9. Fineness CDF on (a) fully explicitly-labeled, (b) partly explicitly-labeled, and (c) not labeled libraries.

Fig. 9 uses CDF plots to compare the fineness of detectors under three categories of libraries. For the fully explicitly-labeled libraries group, shown in Fig. 9 (a), both curves rise shapely when fineness reaches 0.9. For fully explicitly-labeled libraries, PTV can effectively find the location of the label property, and utilize the label path as the minified pTree. However, there are still 13 out of 33 libraries for which $F(PTV)$ is less than 1.0. This occurs because multiple versions have

identical pTrees, which cannot be distinguished using pTree-based methods. This is usually seen with successive minor version updates, where changes are limited to logical changes – no variables are added or removed, and no runtime values of the variables are changed. When this occurs the pTrees between these minor versions are identical. For fully explicit-labeled libraries, having the same pTree for different versions means that the version label in the tree hasn't changed. This occurs for two reasons. First, some sub-versions are labeled with the same major version name. For example, "1.27.0a" and "1.27.0b" will both be labeled as "1.27.0". Second, some versions are simply mislabeled as we discussed in Sec. 5.3. Even for mislabeled versions, PTV can achieve a higher fineness than LDC.

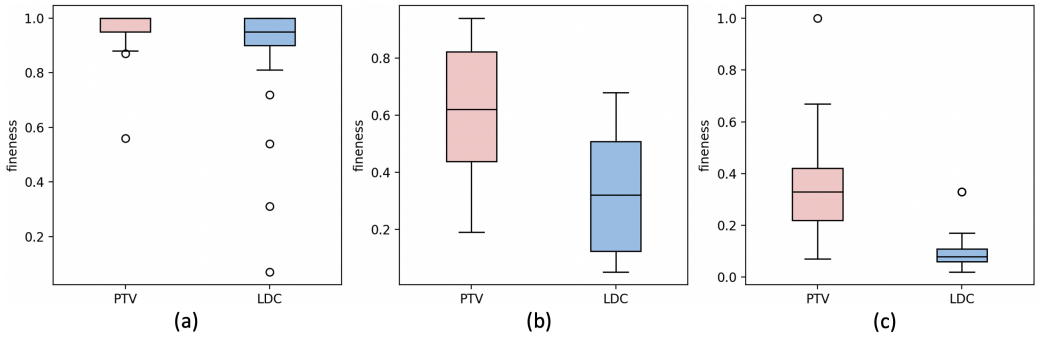

Fig. 10. Fineness distribution of PTV and LDC on (a) fully explicitly-labeled, (b) partly explicitly-labeled, and (c) not labeled libraries.

We show the fineness distribution on fully explicitly-labeled libraries in the box plot Fig. 10 (a). There is one outlier with a very low fineness of PTV – "three.js" whose fineness equals 0.56. "Three.js" is a JavaScript 3D libraries with 248 versions. In Cdnjs repository, most "Three.js" versions have corresponding R versions, like "107" and "R107", "108" and "R108", etc.. After manual inspection, we find that the contents in these pairs are identical, the letter R is just to differentiate the library release location. Versions without R are released on the "Three.js" official website, and R versions are released on its GitHub page. As a result, our tool cannot classify R versions, which leads to a low $F(PTV)$. There are four outliers in the LDC group – "Handlebars" (0.72), "three.js" (0.54), "Hammer.js" (0.31), and "Qooxdoo" (0.07). The "Handlebars" library has a relatively large number of mislabeled versions. The "three.js" library's R versions are also difficult for LDC to identify. The "Hammer.js" library has changed its version label location since the 2.0.0 release and LDC's heuristics do not consider versions before 2.0.0. The situation is more interesting in the case of "Qooxdoo". "Qooxdoo" has a comprehensive and correct version labeling, but LDC fails to find the labeling location since it is too deep in the runtime variable structure of the library, and thus loses the ability to identify the version of this library.

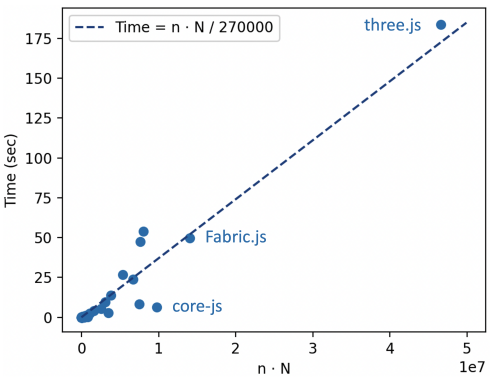

Fig. 11. Time overhead of PTV.

LDC is effective in recognizing explicit-labeled versions, but not for partly explicitly-labeled libraries. Therefore, a clear gap between $F(PTV)$ and $F(LDC)$ can be seen in the Fig. 9 (b) and Fig. 10 (b). For not explicitly-labeled libraries, LDC cannot detect the version and gives an "unknown" result for every version. So, $F(LDC) = 1/|V|$. PTV has a better classification ability for unlabeled versions given in Fig. 9 (c) and Fig. 10 (c), with an average fineness of 0.38, which is higher than the LDC's 0.11. Intuitively, PTV is able to divide every ten none-explicit-labeled versions into four groups on average.

Moreover, we calculate the difference of fineness of two detectors, i.e., $F(PTV) - F(LDC)$. The results show the average value is 0.13. $F(PTV) - F(LDC) = 0$ for 18 libraries; for other libraries, $F(PTV) - F(LDC) > 0$, which means that for each library, the classification ability of PTV will not be less than that of LDC.

RQ3 Conclusion: PTV outperforms LDC in terms of fineness frequency distribution on 64 libraries. For each library, the classification ability of PTV will not be less than that of LDC.

5.5 RQ4: What's the time overhead of pTree minification PTV?

Table 4 shows the time overhead breakdown of each algorithm stage during the pTree processing. In total, generating minified pTrees and their minified supertree sets for 64 libraries takes 1260.3 seconds. The path coloring stage takes up the vast majority of the time (96.66%), and the equivalence class calculation stage takes only 2.89% of the time.

Table 4. Time overhead statistics.

	Equivalence	Path Coloring	Get $\mathbb{S}$	Min Set Cover	Get T_m	Get $\mathbb{S}_m$	Total
Refer	[Jayapaul et al. 2015] Theorem 1	Algo. 1 line 2	Algo. 1 line 3	Algo. 1 line 4-5	Algo. 1 line 6	Algo. 5	-
Time	36.5 s	1218.2 s	1.2 s	0.4 s	0.03 s	3.0 ms	1260.3 s
avg. Time	0.6 s	19.0 s	23.1 ms	5.6 ms	0.4 ms	0.04 ms	19.7 s
Percentage	2.89%	96.66%	0.10%	0.03%	<0.01%	<0.01%	100%

Fig. 11 shows the relationship between the overhead of each library with respect to $n \cdot N$ in the form of a scatter plot, where n is the number of versions, and N is the total number of vertices of generated pTrees. Most of the points in the plot are distributed around the straight line $Time = n \cdot N/270000$. There is only one library that does not adhere to this linear trend – "D3", whose $n \cdot N$ is close to "three.js", but has a high time overhead of 814 seconds. This is the only library whose runtime complexity is exponential.

RQ4 Conclusion: Although the worst-case time complexity for pTree processing is $O(n \cdot N^2)$, the time overhead in practice is close to $\Theta(n \cdot N)$.

6 RELATED STUDY

Forest Algorithm. Trees and forests have been extensively studied. Prior work mainly focus on mining frequent subtrees from databases of labeled trees. To the best of our knowledge, we are first to focus on the problem of finding the unique structure of each tree in the forest and apply it to a real-world detection task. Here we list some key prior works. [Zaki 2002] developed the *TreeMiner* algorithm for mining frequent ordered embedded subtrees. [Asai et al. 2004] proposed the *FREQT* algorithm, which uses an extension-only approach to find all frequent induced subtrees in a database of one or more rooted ordered trees. [Asai et al. 2003; Nakano and Uno 2003] extended

to general case that siblings may have the same labels. [Wang and Liu 1998; Xiao and Yao 2003] first applied path join approach to the mining. [Chi et al. 2003] introduced the *FreeTreeMiner* which applies mining to labeled free trees, which is extended by [Rückert and Kramer 2004]. [Chi et al. 2005] gave a systematic overview of works in this field.

Library Detection. Library detection aims to find the code reuse in software. PTDETECTOR is the first tool proposed for web applications. Prior to it, many approaches have been proposed to detect third-party libraries for desktop and Android applications. The common strategy is extracting features from the source code and matching the binary program library. Binary Analysis Tool (BAT) [Hemel et al. 2011] is a representative binary matching method that utilizes constant values as the detection feature. OSSPolice [Duan et al. 2017] introduces a hierarchical indexing scheme to use better the constant information and the sources' directory tree. Then BCFinder [Tang et al. 2018] makes the indexing lightweight and the detection platform-independent. B2SFinder [Yuan et al. 2019] synthesizes both constant and control-flow features from binary based on their importance-weighting methods to achieve reliable results. Xian Zhan et al. [Zhan et al. 2020] conducted the first empirical study on Android library detection techniques and proposed tool selection suggestions.

Web Library Analysis. Many different kinds of library analysis works have been done. [Feldthaus and Møller 2014] present a pragmatic approach to check the correctness of TypeScript files with respect to JavaScript library implementations. [Kristensen and Møller 2019] explore the concept of a reasonably-most general client and introduce a new static analysis tool for TypeScript verification. [Patra et al. 2018] present an automated method to detect JavaScript libraries' conflicts and show that one out of four libraries is potentially conflicting. [Møller et al. 2020] develop the tool Tapir that finds the relevant locations in the client code to help clients adapt their code to the breaking changes. [Wyss et al. 2022] propose a tool to programmatically detect hidden clones in npm and match them to their source packages. Their tool utilizes a directory tree as a detection feature, which does not apply to the front-end library.

7 DISCUSSION AND CONCLUSION

To enable pTree-based JavaScript library version detection, this paper introduces an algorithm to extract unique feature out of each tree in the forest of pTrees, one for each version. This significantly reduces the space required for version detection. However, the algorithm proposed in this paper is not limited to library version detection; in fact, this algorithm will be a handy tool for any detection problem whose feature can be represented as a tree structure. Compared to widely-used machine-learning-based detection methods, our algorithm provides more fine-grained [8] detection and the extracted feature information is human-readable, which can help developers gain insight into the unique feature patterns of each tree.

Although PTV shows satisfactory performance on minification and detection, there still exist limitations. The first limitation is extensibility. JavaScript libraries are constantly updated. When a new version is developed, the minified pTrees of the whole library need to be recalculated. Another limitation is that the detection result of PTV is only sound when the detection dataset is a subset of the tree processing dataset. In other words, if a library version on the web page is not collected during the pTree generation stage, then PTV may produce unsound results. Thus, using our tool requires a comprehensive collection of all versions of a library.

REFERENCES

Tatsuya Asai, Kenji Abe, Shinji Kawasoe, Hiroshi Sakamoto, Hiroki Arimura, and Setsuo Arikawa. 2004. Efficient substructure discovery from large semi-structured data. *IEICE TRANSACTIONS on Information and Systems* 87, 12 (2004), 2754–2763.

[8] Machine learning methods are hard to achieve one-to-one matching accuracy, and version detection is just such a problem.

Tatsuya Asai, Hiroki Arimura, Takeaki Uno, and Shin-Ichi Nakano. 2003. Discovering frequent substructures in large unordered trees. In *Discovery Science: 6th International Conference, DS 2003, Sapporo, Japan, October 17-19, 2003. Proceedings 6*. Springer, 47–61.

Yun Chi, Richard R Muntz, Siegfried Nijssen, and Joost N Kok. 2005. Frequent subtree mining–an overview. *Fundamenta Informaticae* 66, 1-2 (2005), 161–198.

Yun Chi, Yirong Yang, and Richard R Muntz. 2003. Indexing and mining free trees. In *Third IEEE International Conference on Data Mining*. IEEE, 509–512.

Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*. 2169–2185.

Asger Feldthaus and Anders Møller. 2014. Checking correctness of TypeScript interfaces for JavaScript libraries. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. 1–16.

GitHub. 2023a. johnmichel/Library-Detector-for-Chrome. https://github.com/johnmichel/Library-Detector-for-Chrome/.

GitHub. 2023b. PTV GitHub Homepage. https://anonymous.4open.science/r/PTVgen-5760.

Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. 2011. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. 63–72.

Varunkumar Jayapaul, J Ian Munro, Venkatesh Raman, and Srinivasa Rao Satti. 2015. Sorting and selection with equality comparisons. In *Workshop on Algorithms and Data Structures*. Springer, 434–445.

Tim Kadlec. 2017. 77% of Sites Use at least One Vulnerable JavaScript Library. https://snyk.io/blog/77-percent-of-sites-use-vulnerable-js-libraries/.

Erik Krogh Kristensen and Anders Møller. 2019. Reasonably-most-general clients for JavaScript library analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 83–93.

Xinyue Liu and Lukasz Ziarek. 2023. PTDETECTOR: An Automated JavaScript Front-end Library Detector. In *38th International Conference on Automated Software Engineering*. IEEE/ACM.

Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. 2020. Detecting locations in JavaScript programs affected by breaking library changes. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–25.

Shin-ichi Nakano and Takeaki Uno. 2003. A simple constant time enumeration algorithm for free trees. *PSJ SIGNotes ALgorithms* 091-002 (2003).

Jibesh Patra, Pooja N Dixit, and Michael Pradel. 2018. Conflictjs: finding and understanding conflicts between javascript libraries. In *Proceedings of the 40th International Conference on Software Engineering*. 741–751.

Ulrich Rückert and Stefan Kramer. 2004. Frequent free tree discovery in graph data. In *Proceedings of the 2004 ACM symposium on Applied computing*. 564–570.

Chrome Extension Store. 2023. Library Detector | Developer Tool. https://chrome.google.com/webstore/detail/library-detector/cgaocdmhkmfnkdkbnckgmpopcbpaaejo.

Kwangwon Sun and Sukyoung Ryu. 2017. Analysis of JavaScript programs: Challenges and research trends. *ACM Computing Surveys (CSUR)* 50, 4 (2017), 1–34.

Wei Tang, Du Chen, and Ping Luo. 2018. Bcfinder: A lightweight and platform-independent tool to find third-party components in binaries. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 288–297.

Ke Wang and Huiqing Liu. 1998. Discovering typical structures of documents: A road map approach. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*. 146–154.

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*. 2415–2426.

Yongqiao Xiao and J-F Yao. 2003. Efficient data mining for maximal frequent subtrees. In *Third IEEE International Conference on Data Mining*. IEEE, 379–386.

Zimu Yuan, Muyue Feng, Feng Li, Gu Ban, Yang Xiao, Shiyang Wang, Qian Tang, He Su, Chendong Yu, Jiahuan Xu, et al. 2019. B2SFinder: Detecting open-source software reuse in COTS software. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1038–1049.

Mohammed J Zaki. 2002. Efficiently mining frequent trees in a forest. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. 71–80.

Xian Zhan, Lingling Fan, Tianming Liu, Sen Chen, Li Li, Haoyu Wang, Yifei Xu, Xiapu Luo, and Yang Liu. 2020. Automated third-party library detection for android applications: Are we there yet?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 919–930.