PTV: Better Version Detection of JavaScript Web Libraries Based on Unique Subtree Mining

1st Xinyue Liu
dept. Computer Science and Engineering
University at Buffalo
Buffalo, USA
xliu234@buffalo.edu

2nd Lukasz Ziarek
dept. Computer Science and Engineering
University at Buffalo
Buffalo, USA
lziarek@buffalo.edu

***Abstract*—Identifying the versions of libraries used by a web page is an important task for sales intelligence, website profiling, and even security analysis, enabling more fine-grained web analysis. Recent work uses tree structure to represent the property relationships of the library at runtime, leading to more accurate library identification. However, current 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 minimized. We show that an implementation of our algorithm in a state-of-the-art library detection tool, not only guarantees the soundness of detection result but reduces associated costs where tree-based detection methods can be used to detect library versions. Our experiment results on over 500 real-world libraries with 30,000 unique versions show that our tool reduces space requirements by up to 99% and achieves more precise version detection compared with existing tools.**

***Index Terms*—JavaScript Library, Dynamic Analysis, Version Detection, Tree Algorithm**

I. 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[1]. 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 [2]. 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 front-end 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 versions 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 123 libraries. The most accurate detector, PTDETECTOR [3], 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 built on top of PTDETECTOR. PTV can detect 556 libraries with 30,810 versions, and is anonymously and publicly available here [4]. Our paper makes the following contributions:

1) a novel algorithm to minimize 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. 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 minimize every tree without affecting the detection ability of tree-based library detectors. The tool is published on Google Web Store [5].
3) a comprehensive evaluation of the version detection ability of PTV against existing library detection methods on over 500 real-world libraries with over 30,000 versions. Our results show that PTV reduces the memory footprint by 99.32% without affecting the detection accuracy. In addition, the detection results given by PTV guarantee soundness and are more precise than existing methods.

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

The correctness of the algorithm is proved and the time complexity is analyzed in the technical report submitted in supplementary material.

II. Background and Motivation

A. 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].

B. 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. II-B1 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 [3]. 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. II-B2.

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 [6] and 10,000+ users on the Chrome Extension Store [7]. As a

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

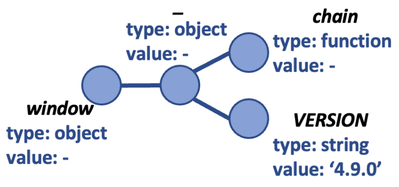

Fig. 1. pTree illustration of Lodash.

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 [3]. 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 versions 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 is infeasible. Over six thousand libraries are registered on Cdnjs, but LDC can support version detection for only 123 libraries. In addition, as we will show in our experiments (Sec. V-D), 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 the libraries that contain version labels have incorrect labels!

2) PTdetector: PTdetector [3] 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 several advantages over traditional methods, however, it does not support version detection.

C. 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.

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. III-B, we will reason about supertrees and introduce an algorithm to correctly identify version.

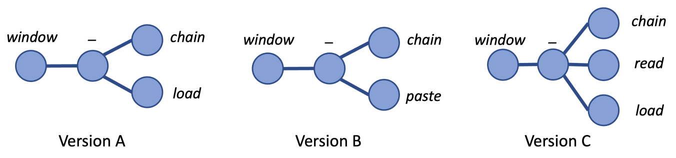

Fig. 2. Example of pTrees of different versions of Lodash. The type and value of each vertex are omitted in the figure. Assume that any vertices with the same name have the same type and value as well.

2) 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 size limit as 50, then over 8G space is needed to store all pTrees. Unfortunately, even a pTree with maximum 50 vertices is not enough to distinguish the subtle differences between the versions.

Our insight is to extract the 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 subtree*. Following this intuition, we are able to design a method to minimize every pTree without affecting the detection ability. In Sec. III-C, we will present the algorithm to find the unique subtree of each tree.

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

III. Algorithm Design

In this section, we describe the core algorithms needed for JavaScript library version detection. Sec. III-A gives basic definitions. Sec. III-B and Sec. III-C provide the solutions to the two challenges introduced in Sec. II-C respectively. A short complexity analysis is given in Sec. III-D. The complete proofs of correctness and complexity analysis of the algorithms are not presented here due to space constraints. We provide them in a technical report for interested readers in the submission supplementary material.

A. Basic Definition

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* $L : V \to \Sigma$. The *size* of T is the number of vertices in the tree.

A *path* is a sequence of vertices $p = (v_1, v_2, ..., v_n) \in V^n$ 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.

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 is a *supertree* of 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. For simplicity, all occurrences of "subtree" in the latter text refer to the induced subtree.

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, it is easy to see 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 $T_1 \neq T_2$.

B. Supertree Exclusion

For a library with n versions, we use the label tree set $\Gamma = \{T_1, T_2, ..., T_n\}$ to represent pTrees for each version. The label tree is used because each vertex in the pTree will carry extra information – name, value, and type – which are represented as labels mapping to vertices.

For a given library loaded at runtime we have a pTree represented by the label tree ϕ. A simple strategy to determine the version of a loaded library is to iterate through trees in Γ and check whether they are subtrees of ϕ. If a given tree is not a subtree, meaning that the web page runtime does not contain the complete pTree information of this library version, then the version corresponding to this tree is not the correct one.

If we find one tree in Γ that is a subtree of ϕ, however, we still can not immediately conclude the version. Assume tree T is a subtree of ϕ, then according to the transitivity of relation $\preceq$, all trees in Γ that are subtrees of T are also subtrees of ϕ. In real-world libraries, the relation $\preceq$ between pTrees from different versions is frequent. This occurs because the action of adding variables and methods in a JavaScript program when updating the version is reflected in the pTree by adding vertices to the original tree. Thus, the old pTree is a subtree of the new one. As a result, when we find one tree is a subtree of ϕ, it is essential to further make sure all the supertrees of this tree are not subtrees of ϕ. Based on this observation, we construct the version detection algorithm shown in Algo. 1.

Before diving into the algorithm, two new definitions need to be introduced to help in its formalization. First, we use the symbol $\mathbb{S}(T)$ to represent the set of all supertrees of T contained in Γ, named *supertree set*. In other words, $\mathbb{S}(T) = \{T' \in \Gamma \mid T \preceq T'\}$. Second, 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 $[T] = \{T' \in \Gamma \,|\, T' = T\}$. Both supertree set and equivalence class can be calculated through trivial tree comparison. Fig. 3 is an example of these two definitions.

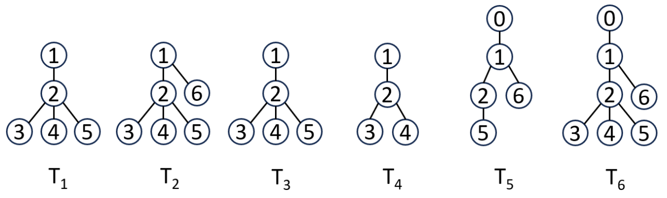

Fig. 3. Assume Γ consists of six trees in the plot, we have the T_1's supertree set $\mathbb{S}(T_1) = \{T_1, T_2, T_3, T_6\}$, and equivalence class $[T_1] = \{T_1, T_3\}$

Algo. 1 shows the algorithm to determine the library version during web page runtime. The inputs are labeled trees set Γ, web runtime pTree ϕ, together with supertree set and equivalence class for each tree in Γ. The algorithm iterates through pTrees in Γ to check whether one of them is a subtree of ϕ (line 2). If so, then check whether all supertrees of this pTree are not subtrees of ϕ (lines 3-7). If so again, return the equivalence class of this tree as the algorithm output (line 8). Here algorithm returns $[T]$ instead of a single tree T because the pTree-based detection algorithm is not able to distinguish between versions whose pTree are equivalent.

C. Unique Subtree Mining

1) Goal: Although we have given a deterministic algorithm to find the base tree, in our practical application scenarios, the library version pTrees (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 minimize the size of trees in Γ by unique subtree mining and ensure that the previous algorithm is still valid. Formally put, given $\Gamma = \{T_1, T_2, ..., T_n\}$, we define its minimized label trees set $\Gamma_m = \{M_1, M_2, ..., M_n\}$, where $M_1 \preceq T_1, M_2 \preceq T_2, \cdots, M_n \preceq T_n$. Our goal is to find a minimum[5] Γ_m that satisfies replacing Γ with this new Γ_m in the input to Algo. 1 will *not* change the algorithm output, i.e., Algorithm1(Γ, ϕ) = Algorithm1(Γ_m, ϕ).

2) Observations: For a tree $T \in \Gamma$, suppose t is a subtree of T (t is not required to be contained in Γ), we say t is an *unique subtree* of T if it is not a subtree of other trees in Γ, i.e., $\forall\, T' \in \Gamma - \{T\}, t \npreceq T'$. Consider that t only appears in the structure of T, so the existence of t during the version detection process indicates the existence of T. As a result, our strategy is to calculate the minimum unique subtree for each tree in Γ, and use these unique subtrees to constitute the new label trees set Γ_m. In other words, for a tree T_i in Γ, we choose its minimum unique subtree as M_i in Γ_m. The uniqueness property of these subtrees ensures the detection algorithm output is unchanged. However, it is easy to induce that for any tree T which has a supertree other than itself, it does not exist unique subtree, because any subtree of T is also a subtree of T's supertrees. As a result, in all subsequent discussions of unique subtree, supertrees are excluded. It is safe to do so because supertrees are also excluded in Algo. 1.

To find the unique subtree, we define the mapping $Rec : p \to \mathcal{P}(\Gamma)$ to record the occurrences of paths in other trees. Concretely speaking, for a path p of tree $T \in \Gamma$, $Rec(p)$ maps to the set of all trees in $\Gamma - \mathbb{S}(T)$ which contain the same path p. Namely, $Rec(p) = \{T' \in \Gamma - \mathbb{S}(T) \,|\, p \in T'.P\}$. In addition, we use the symbol $\mathbb{R}(T)$ to represent the collection of Rec values of all full paths in tree T. In other words, $\mathbb{R}(T) = \{Rec(p) \mid p \in T.P_f\}$. Notice that $\mathbb{R}(T)$ is a multiset because

Algorithm 1 Determine Library Version

```
Input: library version pTrees set $\Gamma$, web runtime pTree $\phi$,
    $\mathbb{S}(T)$ and $[T]$ for each tree $T \in \Gamma$
Output: possible pTrees loaded in $\phi$
 1: for each $T \in \Gamma$ do
 2:   if $T \preceq \phi$ then
 3:     for each $T' \in \mathbb{S}(T)$ do
 4:       if $T' \preceq \phi$ then
 5:         go to 9
 6:       end if
 7:     end for
 8:     return $[T]$
 9:   end if
10: end for
```

[5]The word "minimum" here means the number of all vertices in Γ_m is minimum.

different paths in a tree may have the same Rec value.

Take the tree T_1 in Fig. 3 as an example to illustrate the definition of Rec and $\mathbb{R}$. The tree T_1 has the following three full paths – (1,2,3), (1,2,4), and (1,2,5). For each full path, we check its occurrences in $\Gamma - \mathbb{S}(T_1) = \{T_4, T_5\}$. Observed that the path (1,2,3) only appears in T_4, we can get $Rec((1,2,3)) = \{T_4\}$. Similarly, $Rec((1,2,4)) = \{T_4\}$ and $Rec((1,2,5)) = \{T_5\}$. Finally, we have $\mathbb{R}(T_1) = \{\{T_4\},\{T_4\},\{T_5\}\}$.

Now we give a key proposition about the Rec collection $\mathbb{R}$.

Proposition III-C.1: For any tree $T \in \Gamma$, $\cap\mathbb{R}(T) = \varnothing$.

PROOF. Suppose $\cap\mathbb{R}(T)$ is not an empty set, then there is at least one tree T' in Γ satisfying $T' \in \cap\mathbb{R}(T)$, which means that all the full paths of T also occur in T'. Hence, T' is a supertree of T, i.e., $T' \in \mathbb{S}(T)$. This is contradictory to the definition of Rec, where we exclude the path recording in $\mathbb{S}(T)$. So $\cap\mathbb{R}(T) = \varnothing$. □

Prop. III-C.1 shows that the full paths in tree T will not appear together in any single tree contained in $\Gamma - \mathbb{S}(T)$. In other words, T is a unique subtree of itself. To take it a step further, if we can find a subset $C \subseteq \mathbb{R}(T)$ which still holds $\cap C = \varnothing$, then the tree constructed from the paths in C is a unique subtree of T. Taking the T_1 in Fig. 3 as an example, $C = \{\{T_4\},\{T_5\}\}$ is the smallest subset of $\mathbb{R}(T_1)$ that satisfies $\cap C = \varnothing$. Then we can construct the minimum unique subtree of T_1 by combining a path with Rec value of $\{T_4\}$ and a path with Rec value of $\{T_5\}$. As shown in Fig. 4, the first subtree of T_1 is the combination of path (1, 2, 3) and (1, 2, 4); the second one is the combination of path (1, 2, 3) and (1, 2, 5). Both of them are unique – not subtrees of any tree in $\Gamma - \mathbb{S}(T_1) = \{T_4, T_5\}$.

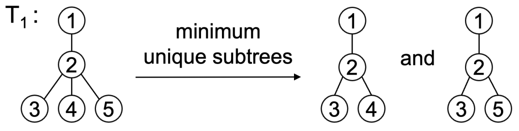

Fig. 4. Tree T_1 in Fig. 3 has two minimum unique subtrees.

Finding the smallest subset C whose intersection is empty 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.

The set cover problem can be solved within approximate polynomial time by a famous greedy algorithm shown in Algo. 2. 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:

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

Algorithm 2 MinCoverSet

```
Input: a set collection: $S = \{\omega_1, \omega_2, ..., \omega_n\}$, where $\omega_i \subseteq \Gamma$
Output: a set $I \subseteq \{1, 2, ..., n\}$, such that $\bigcup_{i\in I} \omega_i = \cup S$
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
```

3) The algorithm: In the previous section, we introduce three new concepts: unique subtree, path record mapping Rec, and record collection $\mathbb{R}$. We elaborate their relationship and then provide a greedy algorithm to calculate an approximated smallest subset C of $\mathbb{R}$ to satisfy $\cap C = \varnothing$, which can help us generate a minimum unique subtree. This section formalizes our observations into the unique subtree mining algorithm shown in Algo. 3.

Algorithm 3 Unique Subtree Mining

```
Input: the labeled trees set $\Gamma = \{T_1, T_2, ..., T_n\}$
Output: the minimized set $\Gamma_m = \{M_1, M_2, ..., M_n\}$
1: Initialization: $\Gamma_m \leftarrow \varnothing$
2: for each $T_i \in \Gamma$ do
3:     calculate $\mathbb{R}(T_i)$
4:     $I \leftarrow MinCoverSet(\{\Gamma - r \mid r \in \mathbb{R}(T_i)\})$
5:     $M_i \leftarrow BuildTreeFromPath(T_i, I)$
6:     $\Gamma_m \leftarrow \Gamma_m \cup \{M_i\}$
7: end for
```

For each tree in Γ, Algo. 3 first calculates its path record collection $\mathbb{R}$. Then in line 4, Algo. 2 is invoked. Notice that the function input is set $\mathbb{R}$ with each element taking the complement, so that, by De Morgan's laws, finding the smallest subset C of $\mathbb{R}$ converts to the set cover problem. The function returns an index set I. In line 5, the unique subtree M_i is built based on the index set I, and is then appended to the set Γ_m in line 6.

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

Algorithm 4 BuildTreeFromPath

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

If we take trees in Fig. 3 as the input to Algo. 3, the output will be Γ_m constituted by unique subtrees displayed in Fig. 5.

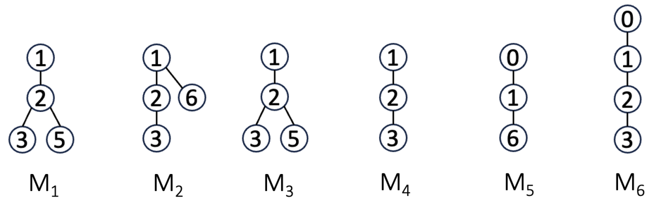

Fig. 5. The minimized label tree set Γ_m, consisting of minimum unique subtrees of trees in Fig. 3.

D. Algorithm Time Complexity

For the unique subtree mining stage, calculating the path record collection $\mathbb{R}$ has the highest complexity. In this step (Algo. 3 line 3), the algorithm needs to check the existence of every full path in all other trees in Γ, so the time complexity is $O(n \cdot N^2)$, where n represents the number of trees in Γ, and N represents the number of vertices in Γ.

When the isomorphism exists between pTrees in large numbers, prioritizing the computation of equivalence classes can effectively decrease time spent on calculating the path record collection $\mathbb{R}$. Prior work [8] 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 the value of n and N in the complexity of path record calculating, and the correctness of the algorithm will not be affected.

The time complexity to determine the library version (Algo. 1) using minimized tree set Γ_m is $O(n)$ because each tree in Γ_m needs to be compared with ϕ at most once. In other words, n times subtree relationship examines is enough to get the algorithm output.

IV. Implementation

We implement our algorithm into a Chrome extension named PTV and published on Google Web Store [5]. Fig. 6 shows the overall workflow of PTV library feature generation and web runtime library version detection. PTV is built on top of PTDETECTOR.

A. Feature Generation Stage

Feature generation stage is completed offline using a trivial local web server. For a library with n versions, first, we load every version of the library file in an empty web page, and use PTDETECTOR to generate the pTree for each library version, represented as $\Gamma = \{T_1, T_2, ..., T_n\}$.

Inner dependency and outer dependency of each version are required as input to PTDETECTOR to eliminate the dependency impact [6]. Outer dependencies can be easily fetched on libraries' official sites, while inner dependencies can only be inferred by reading library raw code, which is time-consuming. However, for 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 on such types 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.

[6]More discussion about inner dependency and outer dependency can be found in [3] Sec.III.C.(1).

Then we use the unique subtree mining algorithm (Algo. 3) to generate the minimized pTrees set Γ_m and save it in a local file for PTV runtime version detection. The original pTree of the library's latest version will be stored for PTDETECTOR library detection.

B. Detection Stage

The detection part of PTDETECTOR is implemented as a Chrome extension that identifies libraries in the browser at runtime. We modify its workflow to enable version detection in PTV as given in the right part of 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. 1 using minimized pTrees set Γ_m as input to identify the specific library version. Another input ϕ to Algo. 1 is the pTree rooted at $\mathcal{X}$. The detected version information will be displayed in the PTV extension popup menu.

V. Evaluation

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

A. Experiment Setup

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 for our tool "in the wild".

B. RQ1: How effective is the minimization of PTV?

To set up an experimental dataset we crawled Cdnjs to gather 700 libraries with the highest GitHub star number. From the top 700 libraries we removed those are not designed to run on the web front-end and those which cannot be loaded successfully due to unknown missing dependencies. We also exclude four frameworks – React, Vue, Next.js, and Preact. As explained in the PTDETECTOR [3], the code for these frameworks mounted on CDN is their runtime debugging tool and we do not consider them in our experiments.

After the exclusions, our dataset consists of 556 libraries with 30,810 versions. We load each on our local server and generate a pTree for each version setting the depth limit as

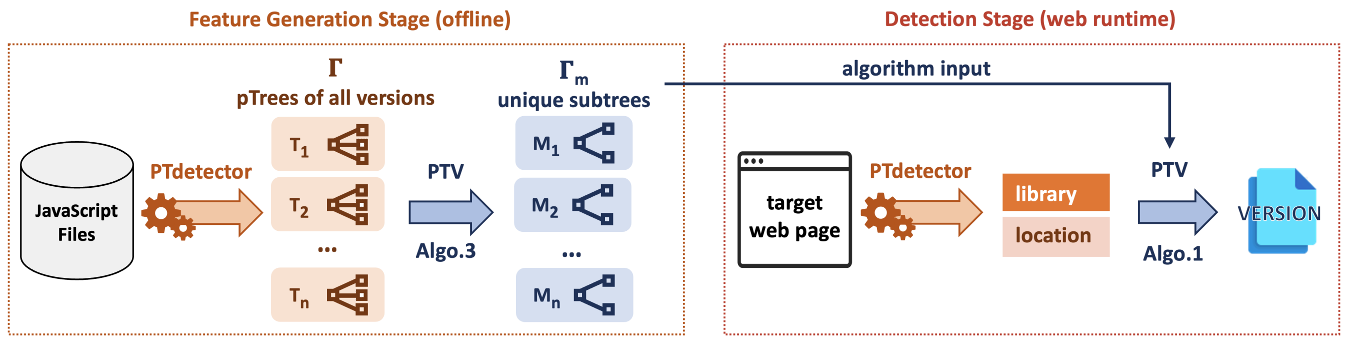

Fig. 6. PTV library version feature generation and runtime detection workflow.

four and the size limit as 1000 [7]. Our result shows that the average size of generated pTrees is 323 so the limit of 1000 is reasonable. After applying the PTV unique subtree mining algorithm, we generate minimized pTrees for every library, with the average size of the pTree being 3.4. Our algorithm reduces the total size of required pTrees for all 556 libraries from 10,654,002 to 72,950. Thus, we are able to reduce the memory footprint by 99.32%. On average, 8 Bytes are required to store one pTree vertex in zipped JSON format. With minimization, to store all pTrees needed for version detection for all libraries currently on Cdnjs, you would need $2,509,859 \times 3.4 \times 8B = 65.45MB$ of space. The detection accuracy is not affected by this reduction as will be shown in subsequent RQs.

Table I shows the time overhead breakdown of each algorithm stage during PTV minimization. In total, generating minimized pTrees for 556 libraries takes 1886.7 seconds (about half an hour), and a single library takes 3.4 seconds on average. Calculating the path record takes up the vast majority of the time (95.0%), and the equivalence class calculation stage takes only 4.9% of the time.

TABLE I
TIME OVERHEAD TO GENERATE FEATURE INFORMATION FOR 556 LIBRARIES.

	Equivalence class	Path Record	Other	Total
Refer	[8] Theorem 1	Algo. 3 line 3	-	-
Time	93.2 s	1791.0 s	2.5 s	1886.7 s
avg. Time	0.2 s	3.2 s	4.5 ms	3.4 s
Percentage	4.9%	95.0%	0.1%	100%

RQ1 Conclusion: PTV greatly reduces the size of the needed pTrees for version detection (99.32%), thus making pTree-based version detection possible. 65 MB is sufficient for all libraries on Cdnjs and the time overhead of PTV minimization workflow is acceptable.

[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 minimization still hold. However, this is not true for the size limit trimming. In practice, we need a size limit to avoid extreme cases.

C. *RQ2: How does* PTV *perform in the wild?*

To answer this question, we test PTV on the 80 top visited websites dataset proposed in the PTDETECTOR paper [3], and compare its detection results with the most popular open-source tool *Library-Detector-for-Chrome (LDC)* and one of the best commercial tools *Wappalyzer*[8]. Wappalyzer has 2,000,000+ users on the Chrome web store. Both LDC and Wappalyzer are hard to automate for testing, so we have to manually open the web pages and record the detection results. To properly measure their version detection ability, some definitions should be introduced in advance.

1) Definitions of measurement: When a library is detected on a web page, detectors will give out a range of versions as the detection result. We use the symbol $\mathcal{D}$ to represent the set of all versions suggested by a detection result (note one detection represents one library). Every element in $\mathcal{D}$ *may* be the true version of this library. Suppose $\mathcal{D}_1$ and $\mathcal{D}_2$ represent the detection result sets of two different tools applied on the same library, depending on the relationship between $\mathcal{D}_1$ and $\mathcal{D}_2$, we specify five relationships shown in Table II to compare the detection ability of the two tools for this library.

TABLE II
FIVE DIFFERENT DETECTION ABILITY RELATIONSHIPS. ($\mathcal{D}_1, \mathcal{D}_2 \neq \varnothing$)

Relationship between $\mathcal{D}_1$ and $\mathcal{D}_2$	Statement
$\mathcal{D}_1 = \mathcal{D}_2$	$\mathcal{D}_1$ and $\mathcal{D}_2$ are *consistent*
$\mathcal{D}_1 \subset \mathcal{D}_2$	$\mathcal{D}_1$ is *more precise* than $\mathcal{D}_2$
$\mathcal{D}_1 \supset \mathcal{D}_2$	$\mathcal{D}_1$ is *less precise* than $\mathcal{D}_2$
$\mathcal{D}_1 \cap \mathcal{D}_2 = \varnothing$	$\mathcal{D}_1$ and $\mathcal{D}_2$ are *inconsistent*
otherwise	$\mathcal{D}_1$ and $\mathcal{D}_2$ are *partly consistent*

We expect that the detection results should be as *precise* as possible. In the best case, there is only one element in the result set – the correct version value. Sometimes the detection results of different tools are *inconsistent* or *partly consistent* if the symmetric difference of result sets is not empty. In such cases, we can not directly compare which tool performs better.

For users, the detection results are normally not shown in the set format, and we need to induce $\mathcal{D}$ based on the result description displayed by the tool. 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” – which are

[8]https://www.wappalyzer.com/

loaded separately into five empty web pages. Then we apply a tool marked A to detect the version of core-js on each web page and collect the detection result. Here, we use $\mathcal{D}_A$ to represent the result set of tool A, and $\mathcal{D}_G$ to represent the ground truth set. Table III demonstrates the value of $\mathcal{D}_A$ under different result descriptions.

TABLE III
AN EXAMPLE TO SHOW HOW TO INDUCE $\mathcal{D}_A$ BASED ON THE DETECTION RESULT DESCRIPTIONS.

$\mathcal{D}_G$	Result description of A	$\mathcal{D}_A$
{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}

As shown in Table III, when the library fails to be detected, $\mathcal{D}_A$ is $\varnothing$; when the detection result is "unknown" for the version but the library is correctly identified, $\mathcal{D}_A$ is the set of all versions, i.e., all versions may be true; other cases follow naturally. Based on the statements in Table II, we can describe the detection ability of A on core-js as: A fails to detect core-js on "2.7.0"; A is less precise than the ground truth on "2.8.0" and "3.0.0"; A is consistent with the ground truth on "2.9.0"; A is inconsistent with the ground truth on "3.1.0".

In some cases, detectors do not provide a result consistent with the ground truth. It is satisfactory enough if the true version is contained in the detection result set, and we call such detection *sound*. Formally put, for one detection on version v, $\mathcal{D}$ is sound if $v \in \mathcal{D}$. Based on this definition, if several tools have inconsistent results in one detection, then at most one of them is sound.

2) Experiment Results: We extend PTDETECTOR to be able to detect 556 libraries (the same used in RQ1) – this system is equivalent to PTV with version detection turned off. Table IV presents the number of detectable libraries, unique detected libraries, and unique detected library occurrences across four tools on the top 80 web pages. We can see that the original PTDETECTOR, which has the feature information of only 83 libraries, shows a similar library detection ability compared with LDC and Wappalyzer. But our extended PTDETECTOR detects 79 different libraries with 413 unique occurrences, almost twice the number of other tools. Furthermore, all the unique library occurrences detected by other tools are also detected by our tool. The occurrence breakdown of each library detected by our tool is shown in Fig. 7.

TABLE IV
NUMBERS OF LIBRARIES DETECTED BY DIFFERENT TOOLS ON THE TOP 80 WEB PAGES.

	LDC	Wappalyzer	PTDETECTOR	**extended PTDETECTOR**
Detectable Libraries	123	unknown	83	**556**
Detected Libraries	32	35	36	**79**
Library Occurrences	238	237	289	**413**

Now we compare the version detection capabilities of existing tools to PTV. Based on the version result descriptions

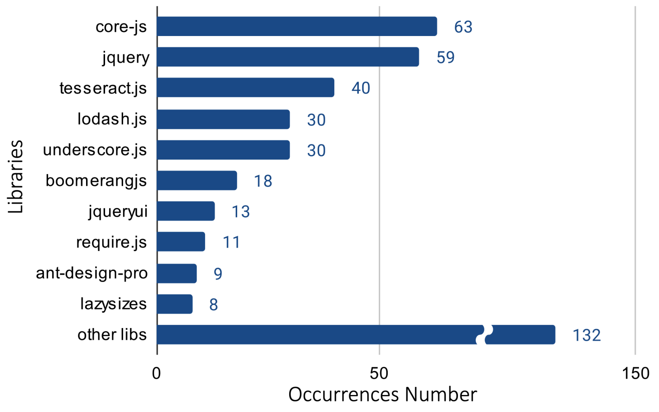

Fig. 7. Library occurrences number detected by extended PTDETECTOR.

given by different tools, for each library occurrence in our experiment, we introduce the version set $\mathcal{D}$ following the definitions in Table III. We then compare the version detection result of PTV against LDC and Wappalyzer respectively (note that we do not compare to PTDETECTOR, as even the extended PTDETECTOR *cannot* detect versions). We count the number of each relationship category proposed in Table II and show the results in the following table (Table V). We do not consider libraries that failed to be identified by LDC or by Wappalyzer for an apples-to-apples comparison.

TABLE V
VERSION DETECTION COMPARISON BETWEEN PTV AND LDC / WAPPALYZER ON THE TOP 80 WEB PAGES.

PTV	Frequency	
	versus LDC	versus Wappalyzer
consistent	228 (95.8%)	192 (81%)
less precise	1 (0.4%)	1 (0.4%)
more precise	3 (1.3%)	31 (13.0%)
inconsistent	6 (2.5%)	13 (5.5%)
partly consistent	0	0
sum	238	237

Table V shows that most version detection results are consistent between all three tools. PTV gives more precise results on 3 occurrences compared to LDC, and on 31 occurrences compared to Wappalyzer. Wappalyzer, despite being a commercial tool, has the lowest precision among the three tools. There is only one occurrence where PTV is less precise providing a range of versions instead of a single version – the library "mustache.js" loaded on the web page *www.dailymail.co.uk*. Library "mustache.js" uses a string variable to store the version information, on which LDC and Wappalyzer based their version detection. Through manual inspection of the library source file, we find that the loaded version is 0.8.2. the library developer forgot to change the value in the version string variable, leaving it as 0.8.1. Although more precise, reporting a single version, LDC and Wappalyzer, nevertheless give an incorrect result identifying the version as 0.8.1. PTV provides a less precise result, giving a range instead of a distinct version: $\mathcal{D}_{PTV} = \{0.8.1, 0.8.2\}$. The pTrees of these two versions are identical and therefore indistinguishable using pTree-based algorithm. However, this result is still sound, since the range does include the *correct* library version.

For the 19 inconsistencies between the tools, we manually verified the version of the library. Verification included checking HTML information, comparing web page JavaScript code, browsing the development blogs of the websites, and investigating the source code of detectors. We conclude that for all inconsistent occurrences, PTV produced sound results. We will reason about why this is the case in RQ3.

Lastly, we examine the runtime overhead of all the tools. For every web page, we record the time starting from clicking the tool button until detection results are displayed. For each tool, we repeat the recording three times and take the average as the overhead value to mitigate the impact of network fluctuations. Fig. 8 uses box plots to depict the overhead distributions of four tools.

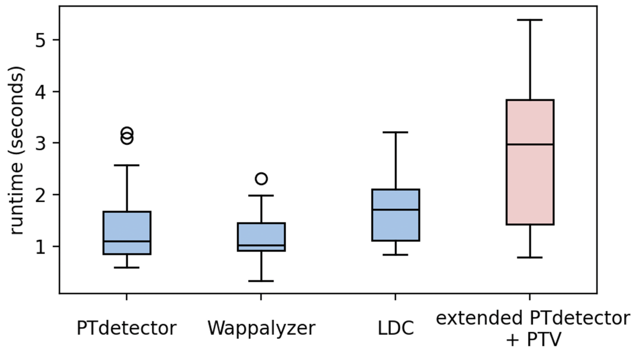

Fig. 8. Runtime overhead distribution of different tools on 80 web pages.

We can observe that Wappalyzer has the fastest response time despite showing the poorest version detection ability. Then comes the original PTDETECTOR, whose response time mostly ranges from 0.95 to 1.75 seconds. The third one is LDC, with a slightly higher response time than PTDETECTOR. Our tool PTV is based on the extended PTDETECTOR, having the highest response time because the number of libraries it integrates is much larger than other tools (556 compared to ∼100). For most of the web pages, our tool can complete detection within five seconds, which is acceptable for average users. In addition, our tool provides an option for users to control the number of libraries they wish to add to the scanning queue, so users can tailor the response time to fit their use cases.

RQ2 Conclusion: Our extended PTDETECTOR can detect far more libraries on web pages. The version detection ability of PTV is consistent with existing tools in a large portion; and is more precise or sound in a small portion.

Besides, although our tool has the highest response time, which is the expense of more detectable libraries, the overall overhead is still within a reasonable range.

D. RQ3: Is PTV *version detection sound?*

1) Minimized pTree Pattern: After manually analyzing minimized 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. Library detectors can determine the existence and version of jQuery by checking the value of this property. We call such property containing version-related information as the *version label*, and the library version with a version label as *explicit-labeled*.

For the explicit-labeled library version, the generated minimized pTree will normally be the exact path of this property containing version information, because the version value is unique among 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 using our tool. Among 556 libraries, we found 98 that have all versions explicit-labeled, 205 that have part of versions explicit-labeled, and the rest have no labeled version. We observed that many libraries don't contain version information in the initial versions, but add it when more versions are developed.

2) Soundness: Determining whether the library detectors are capable of producing correct version detection results is crucial. 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, LDC, and Wappalyzer on the web page. Controlling which version is loaded allows us to establish ground truth. If the detection result does not contain the correct loaded version, we mark this detection as *unsound*. We only consider libraries that can be identified by all three tools – 64 libraries satisfy this requirement. These 64 libraries have 3,533 versions.

The results show that 151 versions are incorrectly identified by LDC; 190 by Wappalyzer; while PTV correctly identifies all 3,533 versions. This is not surprising as PTV guarantees soundness at the algorithm level, i.e., the correct version must be contained in the result. Wappalyzer has more unsound detection than LDC due to uncertain technical defects [9]. For LDC, we find that all incorrect results come from *mislabeling*. As the last research question showed, sometimes library developers forget to update the version property in a newer version. We call such an explicit-labeled version that is assigned with an incorrect version label as a *mislabeled* version. PTV is effective in finding mislabeling. Among the total 2,710 explicit-labeled library versions in the 64 libraries, 151 (5.6%) of them are mislabeled, coming from 23 different libraries. Fig. 9 displays the number of mislabeled versions.

In Fig. 9, most libraries have less than ten mislabeled versions, while libraries "YUI 3" and "FlotCharts" have rather high amounts of mislabeled versions – 37 and 40 respectively. We inspected each of these versions manually. The version

[9]It is hard to reason about this since the source code and the implementation details of Wappalyzer are not publicly available.

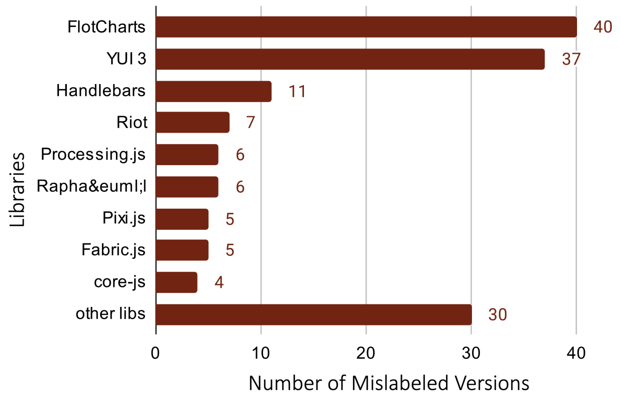

Fig. 9. The mislabeled version number of 23 libraries.

management of both libraries is quite chaotic – more than half of the versions are stored with incorrect version information. Besides, mislabeling appears in both small libraries with less than 4k Github stars – "Rapha&euml;l", "Moment Timezone", "Processing.js", and well-maintained libraries with more than 40k Github stars – "Lo-Dash", "core-js", "Pixi.js". One conclusion is that incorrectly labeled version information is common among web libraries, and determining the version by the version property is not reliable.

Table VI displays the detection result comparison of PTV against two tools and the ground truth $\mathcal{D}_G$ on 64 libraries after excluding unsound results. We can see that to a very large extent (around 99%), the results of PTV are consistent or more precise than LDC and Wappalyzer. There are only a small number of cases where PTV is less precise. These cases are caused by mislabeling. In these PTV will provide less precise but sound results if the pTrees of mislabeled versions are identical. In 70.8% of cases, PTV gives an accurate single version number consistent with the ground truth. In 29.2% of cases, PTV gives a version range as the result, which is less precise than the ground truth but still sound – the correct version is within the identified range.

TABLE VI
VERSION DETECTION COMPARISON BETWEEN PTV AND LDC / WAPPALYZER / GROUND TRUTH ON THE 64 LIBRARIES TEST SUITE. (ONLY CONSIDERING **SOUND** RESULTS)

PTV	Frequency		
	versus LDC	versus Wappalyzer	versus $\mathcal{D}_G$
consistent	2246 (66.0%)	1208 (36.1%)	2503 (70.8%)
less precise	50 (1.5%)	26 (0.8%)	1030 (29.2%)
more precise	1106 (32.5%)	2109 (63.1%)	0
partly consistent	0	0	0
sum	3402	3343	3533

RQ3 Conclusion: PTV is guaranteed to be sound, while LDC and Wappalyzer are not. Among 64 libraries, 23 of them have mislabeled versions leading to unsound detection by LDC and Wappalyzer.

VI. Related Work

Forest Algorithms. 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. [9] developed the *TreeMiner* algorithm for mining frequent ordered embedded subtrees. [10] 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. [11], [12] extended to general case that siblings may have the same labels. [13], [14] first applied path join approach to the mining. [15] introduced the *FreeTreeMiner* which applied mining to labeled free trees, which was extended by [16]. [17] 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) [18] is a representative binary matching method that utilizes constant values as the detection feature. OSSPolice [19] introduced a hierarchical indexing scheme to use better the constant information and the sources' directory tree. Then BCFinder [20] made the indexing lightweight and the detection platform-independent. B2SFinder [21] synthesized both constant and control-flow features from binary based on their importance-weighting methods to achieve reliable results. Xian Zhan et al. [22] 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. [23] presented a pragmatic approach to check the correctness of TypeScript files with respect to JavaScript library implementations. [24] explored the concept of a reasonably-most general client and introduce a new static analysis tool for TypeScript verification. [25] presented an automated method to detect JavaScript libraries' conflicts and showed that one out of four libraries is potentially conflicting. [26] developed the tool Tapir that finds the relevant locations in the client code to help clients adapt their code to the breaking changes. [27] proposed 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.

VII. Discussion and Conclusion

To enable pTree-based JavaScript library version detection, this paper introduces an algorithm to extract unique features out of each tree in the forest of pTrees, one for each version. This significantly reduces the space required for version detection. The algorithm proposed in this paper, however, is not limited to just library version detection. We believe, our algorithm will be a handy tool for any detection problem whose feature can be represented as a tree structure.

Although PTV shows satisfactory performance on tree minimization and detection, there still exist limitations. The first limitation is extensibility. JavaScript libraries are constantly updated. When a new version is developed, the minimized 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. However, considering that our workflow is fully automated, we believe that running the PTV offline feature generation periodically will provide appropriate coverage for newly released libraries.

References

[1] K. Sun and S. Ryu, "Analysis of javascript programs: Challenges and research trends," *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, pp. 1–34, 2017.

[2] T. Kadlec, "77% of sites use at least one vulnerable javascript library," 2017, https://snyk.io/blog/77-percent-of-sites-use-vulnerable-js-libraries/.

[3] X. Liu and L. Ziarek, "Ptdetector: An automated javascript front-end library detector," in *38th International Conference on Automated Software Engineering*. IEEE/ACM, 2023.

[4] GitHub, "Anonymized ptv github homepage," 2024, https://anonymous.4open.science/r/PTV-385B.

[5] "Xxx," 2024, anonymized URL.

[6] GitHub, "johnmichel/library-detector-for-chrome," 2023, https://github.com/johnmichel/Library-Detector-for-Chrome/.

[7] C. E. Store, "Library detector — developer tool," 2023, https://chrome.google.com/webstore/detail/library-detector/cgaocdmhkmfnkdkbnckgmpopcbpaaejo.

[8] V. Jayapaul, J. I. Munro, V. Raman, and S. R. Satti, "Sorting and selection with equality comparisons," in *Workshop on Algorithms and Data Structures*. Springer, 2015, pp. 434–445.

[9] M. J. Zaki, "Efficiently mining frequent trees in a forest," in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2002, pp. 71–80.

[10] T. Asai, K. Abe, S. Kawasoe, H. Sakamoto, H. Arimura, and S. Arikawa, "Efficient substructure discovery from large semi-structured data," *IEICE TRANSACTIONS on Information and Systems*, vol. 87, no. 12, pp. 2754–2763, 2004.

[11] T. Asai, H. Arimura, T. Uno, and S.-I. Nakano, "Discovering frequent substructures in large unordered trees," in *Discovery Science: 6th International Conference, DS 2003, Sapporo, Japan, October 17-19, 2003. Proceedings 6*. Springer, 2003, pp. 47–61.

[12] S.-i. Nakano and T. Uno, "A simple constant time enumeration algorithm for free trees," *PSJ SIGNotes ALgorithms*, no. 091-002, 2003.

[13] K. Wang and H. Liu, "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*, 1998, pp. 146–154.

[14] Y. Xiao and J.-F. Yao, "Efficient data mining for maximal frequent subtrees," in *Third IEEE International Conference on Data Mining*. IEEE, 2003, pp. 379–386.

[15] Y. Chi, Y. Yang, and R. R. Muntz, "Indexing and mining free trees," in *Third IEEE International Conference on Data Mining*. IEEE, 2003, pp. 509–512.

[16] U. Rückert and S. Kramer, "Frequent free tree discovery in graph data," in *Proceedings of the 2004 ACM symposium on Applied computing*, 2004, pp. 564–570.

[17] Y. Chi, R. R. Muntz, S. Nijssen, and J. N. Kok, "Frequent subtree mining–an overview," *Fundamenta Informaticae*, vol. 66, no. 1-2, pp. 161–198, 2005.

[18] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 63–72.

[19] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "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*, 2017, pp. 2169–2185.

[20] W. Tang, D. Chen, and P. Luo, "Bcfinder: A lightweight and platform-independent tool to find third-party components in binaries," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2018, pp. 288–297.

[21] Z. Yuan, M. Feng, F. Li, G. Ban, Y. Xiao, S. Wang, Q. Tang, H. Su, C. Yu, J. Xu *et al.*, "B2sfinder: Detecting open-source software reuse in cots software," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1038–1049.

[22] X. Zhan, L. Fan, T. Liu, S. Chen, L. Li, H. Wang, Y. Xu, X. Luo, and Y. Liu, "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*, 2020, pp. 919–930.

[23] A. Feldthaus and A. Møller, "Checking correctness of typescript interfaces for javascript libraries," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014, pp. 1–16.

[24] E. K. Kristensen and A. Møller, "Reasonably-most-general clients for javascript library analysis," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 83–93.

[25] J. Patra, P. N. Dixit, and M. Pradel, "Conflictjs: finding and understanding conflicts between javascript libraries," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 741–751.

[26] A. Møller, B. B. Nielsen, and M. T. Torp, "Detecting locations in javascript programs affected by breaking library changes," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–25, 2020.

[27] 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*, 2022, pp. 2415–2426.