PTV: Scalable Version Detection of Web Libraries and its Security Application

Xinyue Liu
Chongqing University
Chongqing, China
aaronxyliu@cqu.edu.cn

Haipeng Cai
University at Buffalo
Buffalo, United States
haipengc@buffalo.edu

Lukasz Ziarek
University at Buffalo
Buffalo, United States
lziarek@buffalo.edu

Abstract

Identifying the libraries used by a web application is an important task for sales intelligence, website profiling, and web security analysis. Recent work uses tree structures to represent the property relationships of the library at runtime, realizing automatic library identification without pinpointing versions. But when assessing the security risks associated with these web libraries or conducting fine-grained software analysis, it becomes essential to determine the specific version of the library in use. However, existing tree-based methods are not directly applicable to version detection due to the huge storage requirements for maintaining separate trees for a large number of versions. This paper proposes a novel algorithm to find the most unique structure out of each tree in a forest so that the footprint of the features can be greatly minimized. We implement this algorithm into a web library detection tool. Experimental evaluations on 556 web libraries, encompassing 30,810 versions, reveal that our tool reduces space requirements by up to 99%, achieves more precise version detection compared to existing tools, and detects 190 vulnerabilities on 200 top-traffic websites.

CCS Concepts

• **Software and its engineering → Software testing and debugging**; • **Security and privacy → Web application security**; • **Mathematics of computing → Trees**.

ACM Reference Format:
Xinyue Liu, Haipeng Cai, and Lukasz Ziarek. 2026. PTV: Scalable Version Detection of Web Libraries and its Security Application. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3744916.3773146

1 Introduction

With the increase in the variety of sophisticated web applications, the demand for web 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 web JavaScript libraries[1], almost twice as many as one year ago. With the staggering growth in web libraries, there is an equal need for automatic library detection. Web library techniques are frequently used for competitor analysis, sales intelligence, security analysis, and website profiling.

Version detection is a crucial component in the library detection task, especially with regard to security analysis. Many websites rely on outdated versions of these libraries, which may contain known vulnerabilities. For instance, in 2020, two Cross-Site Scripting (XSS) vulnerabilities[2] were discovered in jQuery versions prior to 3.5.0. These vulnerabilities allowed attackers to inject malicious scripts into web pages, potentially compromising user data or hijacking sessions. Websites using outdated versions of jQuery are exposed to these vulnerabilities, leaving them open to attacks. Many sites were slow to update, either due to a lack of awareness or compatibility concerns – based on the jQuery usage statistics published by BuiltWith[3], about 42% of the sites are still using jQuery versions prior to 3.5.0.

Facing potential security risks posed by web libraries, version detection technique is required to enable organizations to take proactive approaches. By identifying the versions of libraries deployed on websites, they can assess the potential risks and prioritize updates or patches. In addition, many industries are subject to regulatory requirements that mandate the use of secure software components [2, 13, 26]. Version detection techniques provide a means to audit websites and ensure compliance with these standards. Moreover, when a new vulnerability is discovered, security teams can quickly determine how many downstream websites are affected, allowing them to take targeted remediation steps and prevent further exploitation.

Additionally, library version detection has broader applications in facilitating a deeper understanding of the code-level behavior of websites. Accurate program analysis for web applications is widely admitted as a challenging task [17, 28, 35] partly due to the highly dynamic nature of JavaScript and the difficulty of the analysis of prevalent web libraries. Even for the most commonly used library, jQuery, the versions that can be effectively analyzed using traditional methods are limited to versions prior to 2.0.0, a version that was released 11 years ago [3, 18, 27]. Knowing the loaded library version provides new possibilities for analyzing web applications: (1) For static analysis, the behavior of libraries could be separately modeled in advance, thus leading to more reliable static analysis results [35]; (2) For dynamic analysis, version information allows researchers to instrument the correct version of the library, so that information flows can be traced when the library API is invoked.

Although web library detectors exist, determining the version of a detected library remains a non-trivial challenge. Current library

[1]Data source: https://cdnjs.com (Nov. 2024)

ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3773146

[2]CVE-2020-11022 and CVE-2020-11023
[3]jQuery Usage Statistics: https://trends.builtwith.com/javascript/jQuery

detectors depend on manually collecting version patterns for each library, which limits their scalability: the most widely used detector, LDC, is capable of recognizing versions for only 123 libraries. The most accurate detector, PTDetector [21], employs tree structures to automate library feature extraction but faces difficulties in version detection due to the substantial storage requirements for maintaining separate trees for a large number of versions. And this approach results in considerably slow version detection speeds.

In this paper, we follow the idea of using tree structures as the detection feature and propose a novel algorithm – **unique subtree mining** – to minimize trees used in library detection. Our idea is to extract the most unique sub-structure out of each tree in the forest, reducing the content being saved and used for runtime detection. We implemented this algorithm as a tool named **PTV** (shortened for "Pinpointing the Version") to enable tree-based version detection for web JavaScript libraries. PTV can detect 556 libraries with 30,810 versions. We evaluate the version detection capability of PTV against existing methods, and the results demonstrate that PTV reduces the memory footprint by 99.32% and achieves superior detection precision compared to existing tools. Moreover, PTV identifies 190 vulnerabilities across 200 top-traffic websites caused by using outdated libraries, surpassing other tools by 37.7%. In summary, our paper makes the following contributions:

(1) a novel algorithm to mine unique subtrees, which 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 realize web library version detection. The tool is published on the Chrome Web Store [1] and the source code is publicly available on GitHub [15].
(3) a comprehensive evaluation of PTV on 556 real-world libraries and 200 top-traffic websites, where PTV exhibited a more precise version detection capability and identified more vulnerabilities compared to other tools.

2 Background and Motivation

2.1 Web Library

Web libraries are commonly designed to wrap their APIs in objects that are registered in the global context of the browser runtime during the library initialization stage, allowing the APIs to be globally available. In this section, we take Chart.js[4], a commonly used web charting library, as the example. Listing. 1 shows the simplified initialization code of Chart.js (v2.9.3).

```
1  (function() {
2      // Initialize
3      var core_controller = function() {
4          this.construct();
5          return this;
6      };
7      // Define properties
8      core_controller.Animation = ...;
9      core_controller.controllers = ...;
10     core_controller.defaults = ...;
11     ...
12     // Export chart
13     window.Chart = core_controller;
14 }.call(this));
```

Listing 1: Simplified Chart.js Browser Initialization Steps.

In Listing. 1, line 1 defines an anonymous function to wrap all the code, which will execute immediately after its declaration. Line 3 defines the function `core_controller`, which will return an initialized object. Note that a function is also an object in JavaScript. Then, in lines 8 - 11, various APIs (`Animation`, `controller`, `defaults`, and others) are registered as `core_controller` object properties. Finally, in line 13, the `core_controller` object is exposed to the identifier `Chart` in the global context, i.e., registered as a property of `window`[5].

2.2 The Need for Version Detection

One practical application of library version detection is identifying the use of risky outdated libraries on the web. In 2020, a prototype pollution vulnerability was found in Chart.js versions prior to 2.9.4, which is marked as high severity on the Snyk database[6] – it will tamper with the application source code to force the code path that the attacker injects, thereby leading to remote code execution. Based on the library request data provided on jsdeliver[7], Chart.js receives 441,346,805 requests per month from the web, of which 43,769,543 (9.9%) correspond to versions prior to 2.9.4.

Current tools can identify websites that utilize Chart.js; however, they are unable to determine the specific version of it, and as a result, they cannot ascertain whether a website is impacted by this vulnerability. In the subsequent subsections, we will use Chart.js as the detection target to illustrate the existing detection mechanisms, discuss the challenges associated with version identification, and present the insights of our solution.

2.3 Exsiting Detection Methods

Many web JavaScript library detectors exist on the market. Most of them act as browser extensions that detect loaded libraries by checking specific properties at runtime. In Sec. 2.3.1 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 [21]. 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. 2.3.2.

2.3.1 Library-Detector-for-Chrome (LDC). LDC has 600+ stars on GitHub [14] and 10,000+ users on the Chrome Extension Store [34]. As a browser extension, LDC uses dynamic methods to detect libraries. Listing. 2 is the simplified LDC code used to detect Chart.js.

```
1 function testChartjs () {
2     if ( window.Chart ) return true;
3     else return false;
4 }
```

Listing 2: LDC identifies Chart.js by examining whether a property named "Chart" is registered in the global context.

[4]https://www.chartjs.org/

[5]Code running in a web page shares a single global object `window`.
[6]CVE-2020-7746: https://security.snyk.io/vuln/SNYK-JS-CHARTJS-1018716
[7]Chart.js CDN by jsDelivr: https://www.jsdelivr.com/package/npm/chart.js?tab=stats (Data collected in February 2025)

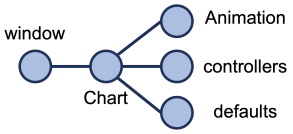

Figure 1: pTree illustration of Chart.js.

Some libraries will store their version tag in a string variable. For example, when LDC detects jQuery, it will read its version directly from the property `window.$.fn.jquery`. We call such property containing version information as the *version label*, and the library version with a version label as *explicit-labeled*. Most detectors on the market today use similar detection methods. However, Chart.js does not provide a version label that can be readily accessed, and this is not an uncommon scenario. We conducted an analysis of the 600 most popular libraries from Cdnjs and found that only 98 of them have all versions explicitly labeled, while 205 have partial version labeling, and the remaining libraries lack any version labeling. Furthermore, as demonstrated in our experiments (Sec. 5.3), version labels, in some cases, even provide incorrect version information.

2.3.2 PTdetector. PTDetector [21] introduces a new concept named *pTree*, which refers to a tree formed by the property relationships between JavaScript variables in a runtime frame. Each vertex in a pTree is assigned 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. This approach has several advantages over traditional methods; however, it does not support version detection.

2.4 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 each 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 the 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 two challenges of this solution in the subsections below.

2.4.1 Correctness. Suppose that Chart.js has only three versions – A, B, and C. Fig. 2 shows the pTrees for these versions. Consider that 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*[8] 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. 3.2, we will reason about supertrees and introduce an algorithm to correctly identify the version.

[8]Similar to a *superset*. The formal definition of supertree will be given in Sec. 3.1

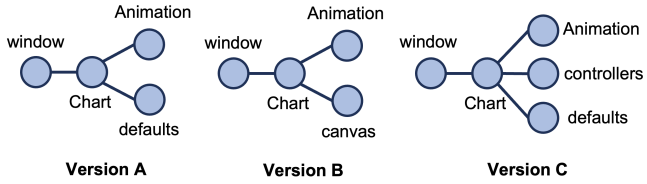

Figure 2: Example of pTrees of different versions of Chart.js.

2.4.2 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 8 GB of space is needed to store all pTrees. Unfortunately, even a pTree with a maximum of 50 vertices is not enough to distinguish the subtle differences between the versions.

Our insight is to extract the most unique structure from each pTree, reducing the content being saved and used for runtime detection. For example, in Fig. 2, we can observe that the property `window.Chart.Animation` appears in all versions; thus, this property does not serve any distinguishing purpose in version detection and should be discarded. In contrast, the property `window.Chart.canvas` only appears in version B and the property `window.Chart.controllers` only appears in version C. Such a property can completely substitute the functionality of the original pTree, being able to uniquely characterize the version. In other words, if the property `window.Chart.canvas` is detected during runtime, we have confidence that the full pTree of version B can be detected. We call such a structure a *unique subtree*. Following this intuition, we are able to design a method to minimize every pTree without affecting the detection ability. In Sec. 3.3, we will present the algorithm to find the unique subtree of each tree.

3 Algorithm Design

In this section, we describe the core algorithms needed for JavaScript library version detection. Sec. 3.1 gives basic definitions. Sec. 3.2 and Sec. 3.3 provide the solutions to the two challenges introduced in Sec. 2.4 respectively. A complexity analysis is given in Sec. 3.4.

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* $L : V \to \Sigma$. 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 $1 \le i < n$. 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 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 $T_2 \preceq T_1$ imply $T_1 = T_2$; $T_1 \preceq T_2$ and $T_2 \preceq T_3$ imply $T_1 \preceq T_3$. We use the symbol $T_1 \prec T_2$ when $T_1 \preceq T_2$ but $T_1 \neq T_2$.

3.2 Supertree Exclusion

For a library with n versions, we use the labeled tree set $\Gamma = \{T_1, T_2, ..., T_n\}$ to represent pTrees for each version. The labeled 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 labeled tree ϕ. A simple strategy to determine the version of a loaded library is to iterate through the 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 cannot 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 that one tree is a subtree of ϕ, it is essential to further ensure that all the supertrees[9] 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, $\mathbb{S}(T) = \{T' \in \Gamma | T \preceq T'\}$. Similarly, we define the *strict supertree set* $\mathbb{S}_{st}(T) = \{T' \in \Gamma \mid T \prec T'\}$. Moreover, we define the *equivalence class* of a tree T with respect to Γ as the set of all trees in Γ that are isomorphic to T, denoted as $[T]$, where $[T] = \{T' \in \Gamma \mid T' = T\}$. Both supertree set and the equivalence class can be calculated through trivial tree comparison. Fig. 3 is an example of these definitions.

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 strict 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 strict 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 pTrees are equivalent.

[9]Strictly speaking, here should be referred to as *strict supertrees* rather than *supertrees*, since every tree is its own supertree.

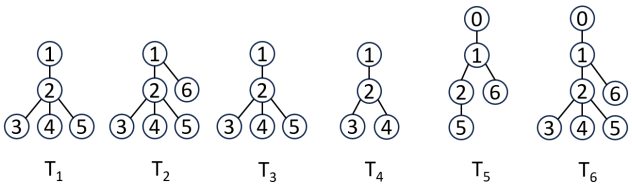

Figure 3: Assume Γ consists of six trees in the plot, we have the T_1's supertree set $\mathbb{S}(T_1) = \{T_1, T_2, T_3, T_6\}$, strict supertree set $\mathbb{S}_{st}(T_1) = \{T_2, T_6\}$, and equivalence class $[T_1] = \{T_1, T_3\}$. Easy to see that $\mathbb{S}(T_1) = \mathbb{S}_{st}(T_1) \cup [T_1]$.

Algorithm 1 Determine Library Version

Input: library version pTrees set Γ, web runtime pTree ϕ, $\mathbb{S}_{st}(T)$ and $[T]$ for each tree $T \in \Gamma$
Output: possible pTrees loaded in ϕ

```
1:  for each T ∈ Γ do
2:    if T ⪯ φ then
3:      for each T' ∈ S_st(T) do
4:        if T' ⪯ φ then
5:          go to 9
6:        end if
7:      end for
8:      return [T]
9:    end if
10: end for
```

3.3 Unique Subtree Mining

3.3.1 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 through unique subtree mining and ensure that the previous algorithm remains valid. Formally put, given $\Gamma = \{T_1, T_2, ..., T_n\}$, we define its minimized labeled 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[10] 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).

3.3.2 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 a *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.

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

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 labeled 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 that 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 a unique subtree, because any subtree of T is also a subtree of T's supertrees. As a result, in all subsequent discussions of unique subtrees, 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 $Rec : p \rightarrow \mathcal{P}(\Gamma)$ 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 $\Gamma - \mathbb{S}(T)$ that contain the same path p. Namely, $Rec(p) = \{T' \in \Gamma - \mathbb{S}(T) \mid p \in T'.P\}$. 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 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 3.3.1. *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. 3.3.1 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, 5); the second one is the combination of path (1, 2, 4) and (1, 2, 5). Both of them are unique – not subtrees of any tree in $\Gamma - \mathbb{S}(T_1) = \{T_4, T_5\}$.

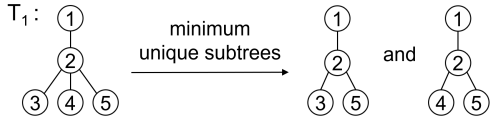

Figure 4: Tree T_1 in Fig. 3 has two minimum unique subtrees.

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 ← ∅, C ← ∅
2: while C ≠ U do
3:     Find the i ∈ {1, 2, ..., n} − I, such that |C ∪ ω_i| is largest
4:     I ← I ∪ {i}
5:     C ← C ∪ ω_i
6: end while
```

3.3.3 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: Γ_m ← ∅
2: for each T_i ∈ Γ do
3:     calculate ℝ(T_i)
4:     I ← MinCoverSet ({Γ − r | r ∈ ℝ(T_i)})
5:     M_i ← BuildTreeFromPath (T_i, I)
6:     Γ_m ← Γ_m ∪ {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.

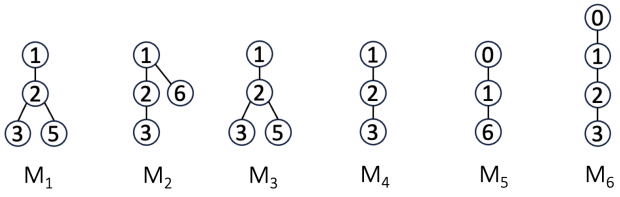

Figure 5: The minimized labeled tree set Γ_m, consisting of minimum unique subtrees of trees in Fig. 3.

3.4 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 [16] 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 calculation, 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 examinations are enough to get the algorithm output.

4 Implementation

We implemented our algorithm into a Chrome extension published on the Chrome Web Store [1]. Fig. 6 shows the overall workflow of PTV library feature generation and web runtime library version detection. PTV is built on top of PTDetector.

4.1 Feature Generation Stage

The feature generation stage is completed offline using a trivial local web server. For a library with n versions, we first 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 [11]. Outer dependencies can be easily fetched from libraries' official sites, while inner dependencies can only be inferred by reading the library's 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 generating large trees. Since this strategy does not consider the actual elements of the data structure it represents, it does not provide effective differentiation among 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.

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.

4.2 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 pTree to identify loaded libraries and their root locations $\mathcal{X}$ in the browser runtime pTree. Then we apply Algo. 1 using the 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.

5 Evaluation

5.1 Experiment Setup

All the experiments are conducted on macOS Sonoma (V 14.1.1) with an Apple M1 chip and 8 GB of memory. All the web pages are opened in Chrome 118.0.5993.88 (Official Build) (arm64).

5.2 RQ1: How effective is the minimization of PTV?

To set up an experimental dataset, we crawled Cdnjs to gather 700 libraries with the highest number of GitHub stars. From the top 700 libraries, we removed those that are not designed to run on the web front-end and those which cannot be loaded successfully due to unknown missing dependencies. We also excluded four frameworks – React, Vue, Next.js, and Preact. As explained in the PTDetector [21], the code for these frameworks mounted on CDN

[11] More discussion about inner dependency and outer dependency can be found in [21] Sec.III.C.(1).

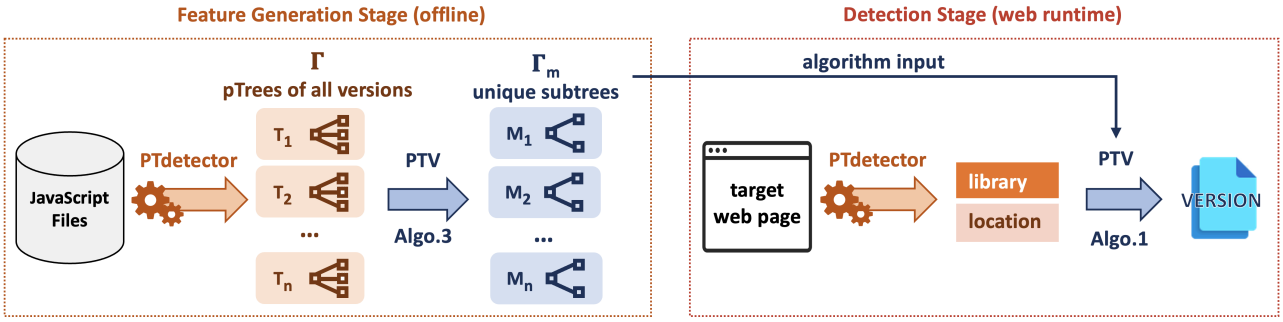

Figure 6: PTV library version feature generation and runtime detection workflow.

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 four and the size limit as 1000 [12]. Our results show that the average size of generated pTrees is 323, so the limit of 1000 is reasonable. During generation, a total of 1,304 (4.2%) library versions reached either the pTree size limit or the depth threshold. 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, $2,509,859 \times 3.4 \times 8B = 65.45MB$ of space is required. The detection precision is not affected by this reduction, as will be shown in subsequent research questions (RQs).

Table 1 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%), while the equivalence class calculation stage takes only 4.9% of the time.

Table 1: Time overhead to generate feature information for 556 libraries.

	Equivalence class	Path Record	Other	Total
Refer	[16] 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%

Answer to RQ1: 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.

[12]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.

5.3 RQ2: Is the result of PTV correct?

In this research question, we apply detectors on our hand-made web pages and compare PTV with the most popular open-source tool *Library-Detector-for-Chrome (LDC)* and one of the best commercial tools *Wappalyzer*[13]. 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.

5.3.1 Definitions of measurement. When a library is detected on a web page, detectors will give out a range of versions as the detection result[14]. 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 2 to compare the detection abilities of the two tools for this library.

Table 2: 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 cannot 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 loaded separately into five empty web pages.

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

[14]PTV gives a range only when there exist isomorphic pTrees of different versions.

Then we apply a tool marked A to detect the version of core-js on each web page and collect the detection results. 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 3 demonstrates the value of $\mathcal{D}_A$ under different result descriptions.

Table 3: 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 3, 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 2, 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 *correct*. Formally put, for one detection on version v, $\mathcal{D}$ is correct 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.

5.3.2 Correctness. Determining whether the PTV are capable of producing correct version detection results is crucial. To test this, we select 64 libraries (encompassing 3,533 versions) that can be version-detected by both LDC and Wappalyzer, set up an empty local web page to load each version of these libraries sequentially, and record the detection results of three tools on this web page. Controlling which version is loaded allows us to establish the ground truth. If the detection result does not contain the correct loaded version, we mark this detection as *incorrect*.

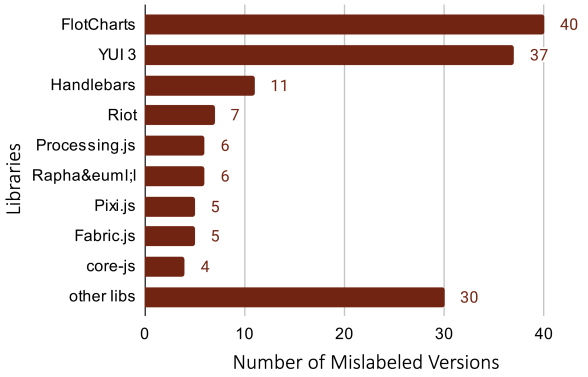

Figure 7: The mislabeled version number of 23 libraries.

Our 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 correct results at the algorithm level, i.e., the correct version must be contained in the result. Wappalyzer has more incorrect detections than LDC due to uncertain technical defects[15]. For LDC, we find that all incorrect results stem from *mislabeling*. Recall that LDC identifies versions by reading labels, but sometimes library developers forget to update the version label in a newer version. We call such an explicit-labeled version that is assigned an incorrect version label a *mislabeled* version. PTV is effective in finding mislabeling. Among the total of 2,710 explicit-labeled library versions in the 64 libraries, 151 (5.6%) of them are mislabeled, coming from 23 different libraries. Fig. 7 displays the number of mislabeled versions.

In Fig. 7, most libraries have fewer than ten mislabeled versions, while libraries "YUI 3" and "FlotCharts" have rather high numbers of mislabeled versions – 37 and 40, respectively. We inspected each of these versions manually. The version 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 – "Raphaël", "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 4: Version detection comparison between PTV and LDC / Wappalyzer / ground truth on the 64 libraries test suite. (Only considering correct 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

Table 4 displays the detection result comparison of PTV against two tools and the ground truth $\mathcal{D}_G$ on 64 libraries after excluding incorrect results. We can see that to a very large extent (around 99%), the results of PTV are consistent with 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 identical pTrees. In these PTV will provide less precise but correct 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. This occurs because some pTrees of different versions are isomorphic. As a result, pTree-based methods are theoretically unable to distinguish between these versions and will output all of them as potential candidates.

Overall, the correctness of our approach is grounded in the theoretical soundness of our detection algorithm. It ensures conservative

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

detection: if multiple versions are indistinguishable, it reports a version range instead of a potentially incorrect single version.

5.3.3 Bundling. To evaluate whether PTV can effectively detect libraries that have been wrapped by bundlers, we conducted a preliminary experiment testing various bundling configurations. We selected ten of the most popular libraries in our dataset—based on GitHub star counts—that have up-to-date counterparts on npm. The latest versions of these libraries were imported from npm, bundled using Webpack[16] (v5.99.9) in "production" mode, and deployed on our minimal test website.

In the first configuration, we explicitly exposed each library to the global scope using the Webpack expose-loader[17]. Under this setting, PTV successfully identified all ten libraries. In contrast, when using the default Webpack configuration — without expose-loader — PTV was able to detect seven out of the ten libraries: *three.js*, *jQuery*, *Lo-Dash*, *Leaflet*, *Backbone*, *Underscore*, and *core-js*. The remaining three libraries — *Bootstrap*, *D3*, and *Moment.js* — were not detected.

By default, Webpack employs Immediately Invoked Function Expressions (IIFEs) to isolate libraries in local scopes, preventing them from leaking into the global namespace. However, we found that the seven detected libraries explicitly register their identifiers on the global `window` object when they find the browser environment. This behavior leaves a detectable memory footprint that PTV can leverage for identification. This design choice aligns with the intended usage model of many web libraries: they are meant to provide globally accessible APIs that remain available throughout the entire lifecycle of a web page to handle user interactions dynamically, rather than acting as temporary variables that are discarded after page initialization.

Answer to RQ2: PTV correctly identified all the library versions in our experiment set, while LDC and Wappalyzer did not. Among 64 libraries, 23 of them have mislabeled versions leading to incorrect detection by LDC and Wappalyzer. Besides, PTV exhibits partial effectiveness on detecting bundled libraries.

5.4 RQ3: How does PTV perform in the wild?

To answer this question, we evaluated PTV using the 200 top-traffic websites dataset introduced in the PTDetector paper [21], and compared its results with those of LDC and Wappalyzer. We aimed to assess PTV's performance in one of its most direct applications: identifying vulnerabilities across these websites.

5.4.1 Library Identification. We extend PTDetector to be able to detect 556 libraries (the same ones used in RQ1) – this system is equivalent to PTV with version detection turned off. Table 5 presents the number of detectable libraries, detected libraries, and detected library occurrences across four tools on the homepages of 200 top websites. We can see that the original PTDetector, which contains the feature information of only 83 libraries, shows a similar library detection ability compared to LDC and Wappalyzer. But our extended PTDetector detects 79 different libraries with 413 occurrences, almost twice the number of other tools. Furthermore, all library occurrences detected by other tools are also detected by our tool. The breakdown of occurrences for each library detected by our tool is shown in Fig. 8.

Table 5: Numbers of libraries detected by different tools on the top 200 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**

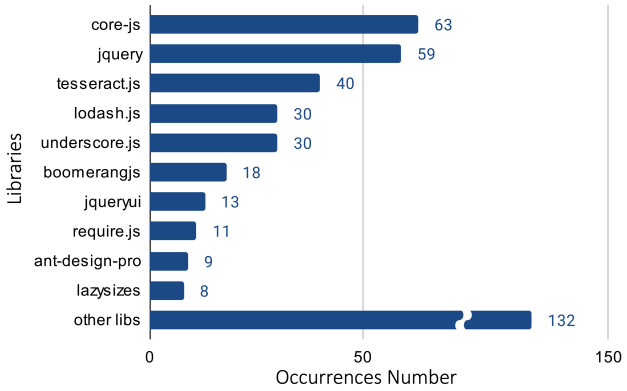

Figure 8: Library occurrences number detected by extended PTDetector.

5.4.2 Vulnerability Assessment. We collected the version detection results from LDC, Wappalyzer, and PTV (built on the extended PTDetector) and cross-referenced them with the Snyk database, which specifies the impacted version ranges of libraries, enabling us to assess the number of vulnerabilities present on these websites. When counting the number of vulnerabilities, we adopt a conservative approach: a vulnerability is considered detected only if the version range identified by the detection tool falls entirely within the impacted version range specified for the vulnerability. Note that we are not confirming whether the vulnerable code has been actively used on the site but are highlighting the presence of vulnerable libraries, which inherently pose a security risk.

Out of the 200 websites analyzed, 77 were found to include at least one library, and more than half of these (44 sites) had at least one vulnerability due to using identified outdated libraries. Table 6 provides a breakdown of the number of known web library vulnerabilities identified by the three tools, categorized into nine vulnerability classes. In summary, PTV detects the highest number of vulnerabilities, encompassing all those identified by LDC and Wappalyzer, and is the only tool that uncovers a critical vulnerability. This vulnerability was found on the U.S. government website www.noaa.gov, which uses an outdated version of the JavaScript template library "handlebars.js"[18]. LDC and Wappalyzer detected

[16]Webpack, a widely-used module bundler for web: https://webpack.js.org/
[17]Webpack expose-loader: https://webpack.js.org/loaders/expose-loader/

[18]CWE-1321: A prototype pollution, affecting handlebars.js versions <3.0.8 or >=4.0.0 <4.5.3. The version used on www.noaa.gov is 4.0.4.

Table 6: Vulnerabilities found by each detector. For each tool we show the severity of the identified vulnerabilities: critical (C), high (H), medium (M), low (L), and their total number (T). The winning numbers of PTV are flagged using gray boxes.

Class	Affected libraries	Vulnerability occurrences found												
		by LDC				by Wappalyzer				by PTV				
		H	M	L	T	H	M	L	T	C	H	M	L	T
Cross-site Scripting (XSS)	14	2	76	19	**97**		77	19	**96**		11	100	24	**135**
Prototype Pollution	6	20	4		**24**	20	4		**24**	1	25	7		**33**
ReDoS	3	2	10		**12**	2	10		**12**		3	10		**13**
Code Injection	1	4			**4**	4			**4**		4			**4**
Content Injection	1		1		**1**		1		**1**			1		**1**
DoS	1				**0**				**0**		1			**1**
Remote Code Execution	1				**0**				**0**		1			**1**
Template Injection	1				**0**				**0**			1		**1**
Arbitrary Code Execution	1				**0**				**0**		1			**1**
TOTAL	19	28	91	19	**138**	26	92	19	**137**	1	46	119	24	**190**

the presence of this library, but they failed to determine its specific version due to the lack of manually collected version patterns, resulting in the oversight of this critical vulnerability; while PTV can pinpoint the version through matching the unique features automatically extracted from each version of the library.

5.4.3 Overhead Analysis. 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. 9 uses box plots to depict the overhead distributions of four tools.

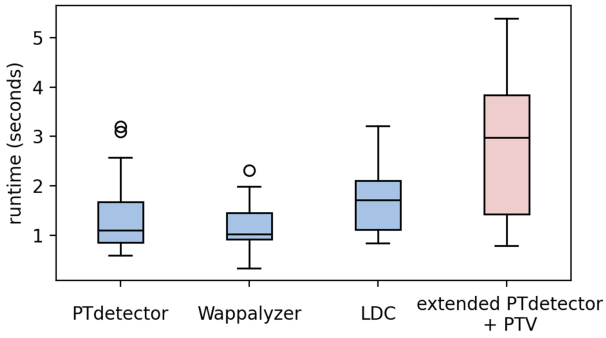

Figure 9: Runtime overhead distribution of different tools on 200 web pages.

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.

Answer to RQ3: Our extended version of PTDetector is capable of detecting significantly more libraries on web pages. Compared to existing tools, PTV identifies 52 (37.7%) additional vulnerabilities while maintaining a reasonable performance overhead.

6 Related Work

Tree and Forest Algorithms. In the program analysis field, tree algorithms are often applied to structures such as abstract syntax trees (ASTs) [7, 10, 12, 32]. These works focus on identifying similarities or differences between two trees, rather than addressing the multiple-to-one matching problem introduced in our paper. Regarding forest algorithms, prior work mainly centered on mining frequent subtrees from databases of labeled trees. To the best of our knowledge, we are the first to address the problem of identifying the unique substructure of each tree in the forest and to apply this approach to a real-world detection task. Here we list some key prior works. [40] developed the *TreeMiner* algorithm for mining frequent ordered embedded subtrees. [4] proposed the *FREQT* algorithm. [5, 24] extended to the general case in which siblings may have the same labels. [36, 39] first applied the path join approach to the mining. [9] introduced the *FreeTreeMiner*, which applied mining to labeled free trees and was extended by [31]. [8] gave a systematic overview of works in this field.

Library Detection. Library detection aims to find the code reuse in software. PTDetector [21] is the first academic tool proposed for web applications. Prior to it, many approaches have been proposed to detect third-party libraries for desktop and Android applications. Various research [6, 20, 22, 25, 33, 37, 42] tried to extract features of libraries and use different techniques to identify libraries and their versions. Xian Zhan et al. [41] conducted the first empirical study on Android library detection techniques. However, all of these methods are static analysis, which cannot identify libraries that are dynamically loaded at runtime or are with dynamic behaviors, which is a common case for web libraries.

Web Library Analysis. Many different kinds of library analysis works have been done. [11] presented a pragmatic approach to check the correctness of TypeScript files with respect to JavaScript library implementations. [19] explored the concept of a reasonably most general client and introduced a new static analysis tool for TypeScript verification. [29] presented an automated method to

detect JavaScript libraries' conflicts and showed that one out of four libraries is potentially conflicting. [23] developed the tool Tapir that finds the relevant locations in the client code to help clients adapt their code to the breaking changes. [38] 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. [30] conducted a large-scale empirical analysis of bundled web libraries and assessed their posture with regards to software supply chain security. However, due to the absence of an advanced version detection tool, they only identified the version for one library — Lodash — resulting in coarse vulnerability analysis.

7 Threats to Validity

The first threat concerns our evaluation in RQ2, where we construct a controlled set of synthetic web pages to isolate and assess the correctness of library version detectors. While this setup enables fine-grained validation on individual versions, it may not fully capture the complexity of real-world deployments – factors such as the presence of library versions released after our feature generation stage may lead to incorrect results when applied to live websites. Moreover, the 64 libraries selected for evaluation are primarily widely used libraries, which tend to have richer pTree representations, allowing PTV to more effectively differentiate versions. For less popular libraries with smaller or less distinctive pTrees, PTV can still produce correct results but probably with reduced precision compared to the performance observed in our experiments.

Another threat to validity is that the RQ3 results based on the top-traffic websites may not be able to be generalized to other websites. Although we believe that the top-traffic websites provide a good overview of the web, our experiment results may not be reproducible on more specialized and niche websites, as they may utilize more specialized libraries.

For the vulnerabilities identified, we did not conduct an in-depth analysis to determine whether they could be practically exploited. This paper primarily focuses on validating the feasibility of the proposed algorithm. A comprehensive security analysis would require substantial additional effort and is therefore left for future work.

8 Limitations

One limitation of our approach is the requirement of retraining – going through the feature generation workflow every time a new library version is released. To mitigate this limitation, we have fully automated the PTV workflow. The cost to generate features from scratch is reasonable — half an hour for 556 libraries. This automation enables PTV to periodically crawl all libraries, generate up-to-date detection features, and ensure coverage of the latest versions. We picked 1 week as the crawling interval, but this can be changed to better mirror the release schedule of libraries.

Another limitation of PTV is its inability to detect all libraries that are wrapped within bundlers, as mentioned in RQ2. To enable pTree-based detection for libraries hidden in local scopes, a promising direction is the use of static instrumentation techniques to explicitly expose library objects to the global context prior to runtime analysis. For example, the method proposed by Rack and Staicu [30] can be employed to locate import statements in the bundled code — such as "`var t = n(692);`" — and inject a line like "`window.scope01 = { t };`" to make the library object globally accessible. This approach can be further integrated as a preprocessing toolchain within PTV.

9 Conclusion

To enable pTree-based web 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.

Conceptually, the JavaScript execution environments across web and npm platforms are similar, both of which support pTree-based analysis. Thus, we believe our tool, with minor modifications, could be applied on the npm platform. Besides, the algorithm proposed in this paper is not limited to just library version detection; we believe our algorithm will be a handy tool for any detection problem whose features can be represented as a tree structure.

Acknowledgments

This work was supported in part by the U.S. National Science Foundation (NSF) under Grants No. 2211997 and CCF-2505223, and by the Office of Naval Research (ONR) under Grant No. N000142512252. The authors thank the anonymous reviewers for their valuable feedback and suggestions.

References

[1] 2025. Library Detector (Academic Tool), the Chrome Web Store–published counterpart of PTV. https://chromewebstore.google.com/detail/library-detector-academic/liedgiagjapaehficeimmjcemnknmdfp.

[2] America's Cyber Defense Agency. 2022. Apache Log4j Vulnerability Guidance. https://www.cisa.gov/news-events/news/apache-log4j-vulnerability-guidance.

[3] Esben Andreasen and Anders Møller. 2014. Determinacy in static analysis for jQuery. *SIGPLAN Not.* 49, 10 (Oct. 2014), 17–31. doi:10.1145/2714064.2660214

[4] 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.

[5] Tatsuya Asai, Hiroki Arimura, Takeaki Uno, and Shin-ichi Nakano. 2003. Discovering Frequent Substructures in Large Unordered Trees. In *Discovery Science*, Gunter Grieser, Yuzuru Tanaka, and Akihiro Yamamoto (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 47–61.

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

[7] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. 368–377. doi:10.1109/ICSM.1998.738528

[8] 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. doi:10.3233/FUN-2005-661-208

[9] Y. Chi, Y. Yang, and R.R. Muntz. 2003. Indexing and mining free trees. In *Third IEEE International Conference on Data Mining*. 509–512. doi:10.1109/ICDM.2003.1250964

[10] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 313–324. doi:10.1145/2642937.2642982

[11] Asger Feldthaus and Anders Møller. 2014. Checking correctness of TypeScript interfaces for JavaScript libraries. *SIGPLAN Not.* 49, 10 (Oct. 2014), 1–16. doi:10.1145/2714064.2660215

[12] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. 2007. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering* 33, 11 (2007), 725–743. doi:10.1109/TSE.2007.70731

[13] Fortinet. 2019. Solar Winds Cyber Attack. https://www.fortinet.com/resources/cyberglossary/solarwinds-cyber-attack.

[14] GitHub. 2023. johnmichel/Library-Detector-for-Chrome. https://github.com/johnmichel/Library-Detector-for-Chrome/.

[15] GitHub. 2025. PTV GitHub Homepage. https://github.com/aaronxyliu/PTVgen.

[16] Varunkumar Jayapaul, J. Ian Munro, Venkatesh Raman, and Srinivasa Rao Satti. 2015. Sorting and Selection with Equality Comparisons. In *Algorithms and Data Structures*, Frank Dehne, Jörg-Rüdiger Sack, and Ulrike Stege (Eds.). Springer International Publishing, Cham, 434–445.

[17] Simon Holm Jensen, Magnus Madsen, and Anders Møller. 2011. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 59–69. doi:10.1145/2025113.2025125

[18] Yoonseok Ko, Xavier Rival, and Sukyoung Ryu. 2017. Weakly Sensitive Analysis for Unbounded Iteration over JavaScript Objects. In *Programming Languages and Systems*, Bor-Yuh Evan Chang (Ed.). Springer International Publishing, Cham, 148–168.

[19] 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)*. 83–93. doi:10.1109/ICSE.2019.00026

[20] Menghao Li, Wei Wang, Pei Wang, Shuai Wang, Dinghao Wu, Jian Liu, Rui Xue, and Wei Huo. 2017. LibD: Scalable and Precise Third-Party Library Detection in Android Markets. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 335–346. doi:10.1109/ICSE.2017.38

[21] Xinyue Liu and Lukasz Ziarek. 2024. PTdetector: An Automated JavaScript Front-End Library Detector. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering* (Echternach, Luxembourg) *(ASE '23)*. IEEE Press, 649–660. doi:10.1109/ASE56229.2023.00049

[22] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: fast and accurate detection of third-party libraries in Android apps. In *Proceedings of the 38th International Conference on Software Engineering Companion* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 653–656. doi:10.1145/2889160.2889178

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

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

[25] Annamalai Narayanan, Lihui Chen, and Chee Keong Chan. 2014. AdDetect: Automated detection of Android ad libraries using semantic analysis. In *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*. 1–6. doi:10.1109/ISSNIP.2014.6827639

[26] U.S. House of Representatives Committee on Oversight and Government Reform. 2018. The Equifax Data Breach Report. https://oversight.house.gov/wp-content/uploads/2018/12/Equifax-Report.pdf.

[27] Changhee Park, Hyeonseung Im, and Sukyoung Ryu. 2016. Precise and scalable static analysis of jQuery using a regular expression domain. *SIGPLAN Not.* 52, 2 (Nov. 2016), 25–36. doi:10.1145/3093334.2989228

[28] Joonyoung Park, Inho Lim, and Sukyoung Ryu. 2016. Battles with false positives in static analysis of JavaScript web applications in the wild. In *Proceedings of the 38th International Conference on Software Engineering Companion* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 61–70. doi:10.1145/2889160.2889227

[29] 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* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 741–751. doi:10.1145/3180155.3180184

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

[31] Ulrich Rückert and Stefan Kramer. 2004. Frequent free tree discovery in graph data. In *Proceedings of the 2004 ACM Symposium on Applied Computing* (Nicosia, Cyprus) *(SAC '04)*. Association for Computing Machinery, New York, NY, USA, 564–570. doi:10.1145/967900.968018

[32] Tobias Sager, Abraham Bernstein, Martin Pinzger, and Christoph Kiefer. 2006. Detecting similar Java classes using tree algorithms. In *Proceedings of the 2006 International Workshop on Mining Software Repositories* (Shanghai, China) *(MSR '06)*. Association for Computing Machinery, New York, NY, USA, 65–71. doi:10.1145/1137983.1138000

[33] Charlie Soh, Hee Beng Kuan Tan, Yauhen Leanidavich Arnatovich, Annamalai Narayanan, and Lipo Wang. 2016. LibSift: Automated Detection of Third-Party Libraries in Android Applications. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. 41–48. doi:10.1109/APSEC.2016.017

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

[35] Kwangwon Sun and Sukyoung Ryu. 2017. Analysis of JavaScript Programs: Challenges and Research Trends. *ACM Comput. Surv.* 50, 4, Article 59 (Aug. 2017), 34 pages. doi:10.1145/3106741

[36] 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.

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

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

[39] Y. Xiao and J.-F. Yao. 2003. Efficient data mining for maximal frequent subtrees. In *Third IEEE International Conference on Data Mining*. 379–386. doi:10.1109/ICDM.2003.1250943

[40] 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* (Edmonton, Alberta, Canada) *(KDD '02)*. Association for Computing Machinery, New York, NY, USA, 71–80. doi:10.1145/775047.775058

[41] Xian Zhan, Lingling Fan, Tianming Liu, Sen Chen, Li Li, Haoyu Wang, Yifei Xu, Xiapu Luo, and Yang Liu. 2021. 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* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 919–930. doi:10.1145/3324884.3416582

[42] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. 2013. Fast, scalable detection of "Piggybacked" mobile applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy* (San Antonio, Texas, USA) *(CODASPY '13)*. Association for Computing Machinery, New York, NY, USA, 185–196. doi:10.1145/2435349.2435377