PTDETECTOR: An Automated JavaScript Front-end Library Detector

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

2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 979-8-3503-2996-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ASE56229.2023.00049

***Abstract*—Identifying what front-end library runs on a web page is challenging. Although many mature detectors exist on the market, they suffer from false positives and the inability to detect libraries bundled by packers such as Webpack. Most importantly, the detection features they use are collected from developers' knowledge leading to an inefficient manual workflow and a large number of libraries that the existing detectors cannot detect. This paper introduces PTDETECTOR, which provides the first automated method for generating features and detecting libraries on web pages. We propose a novel data structure, the *pTree*, which we use as a detection feature. The *pTree* is well-suited for automation and addresses the limitations of existing detectors. We implement PTDETECTOR as a browser extension and test it on 200 top-traffic websites. Our experiments show that PTDETECTOR can identify packer-bundled libraries, and its detection results outperform existing tools.**

I. INTRODUCTION

The rapid growth of the web environment makes simple single-file JavaScript programs a thing of the past. Chrome first supported ES6 modules in 2017 [1]. This browser-native support for multiple file structures shows that JavaScript is no longer the scripting language it was initially designed to be. Instead, it is tasked with the responsibility of developing large, complex web applications. Along with this, many JavaScript libraries have been developed. Cdnjs, the largest CDN (Content Delivery Networks) built to serve websites, contains 4,366 different JavaScript libraries[1]. Based on a technology survey [2], only 17.7% of websites use no JavaScript libraries.

JavaScript front-end library detection is helpful for companies and researchers. From the industry perspective, library detection is used for competitor analysis, sales intelligence, and website profiling. Survey sites, such as W3Techs, will provide these web market research services to companies. Library detection also plays an important role in research. Better library detection enables more detailed web dependency modeling, thus improving the accuracy of web static analysis. Besides, regarding web security, a group of practitioners ran tests on 5,000 top websites and discovered that 76.6% of them include at least one vulnerable library [3]. Library detection techniques can efficiently find libraries with potential risks and provide fixing recommendations.

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

These days, many JavaScript library detectors exist. Some are integrated into commercial web analytic platforms and provide reports as a paid service, while others are open-source Chrome extensions. These tools mainly use two kinds of detection features: globally defined library properties and the library file name. However, our preliminary study indicates that both features can lead to false positives due to their overly simple detection mechanism. Additionally, existing library detectors cannot handle libraries wrapped by packers, such as Webpack, a module bundler that reorganizes the library file structure and wraps the library into the local scope. Many reviews of the detection extensions complain about their inability to detect Webpack-bundled libraries. Worse, these features have to be collected manually using developers' knowledge about the library, which is a labor-intensive and error-prone task. The most popular detectors can recognize no more than 100 libraries to date. Although many concrete research works regarding library automated detection have been proposed for desktop and Android applications, they cannot directly apply to websites due to huge environment differences.

To address these limitations, this paper makes the following contributions:

1) We introduce the first automated JavaScript front-end library detector, PTDETECTOR. Our tool takes JavaScript files and their dependency information as input and automatically extracts detection features using a trivial localhost client. We implement the detection component of PTDETECTOR as a Chrome extension which is available open-source on GitHub [4]. PTDETECTOR requires less than 16MB to detect all libraries on Cdnjs.
2) We present a novel data structure — property tree (*pTree*) — to depict the properties registered by the loaded library. And we use a weight-based tree-matching algorithm to score the existence possibility of libraries on the page. To eliminate the impact of library dependencies and improve performance, a series of algorithms are proposed for pTree post-processing. Compared with the file name and property analysis, the rich details provided by the tree structure allow PTDETECTOR to distinguish libraries more accurately and detect libraries wrapped in local scope by packers.
3) We conduct a real-world study of PTDETECTOR on the

2643-1572/23/$31.00 ©2023 IEEE
DOI 10.1109/ASE56229.2023.00049

200 top-traffic ranking websites and benchmark against the most popular open-source detector, LDC, and the most popular commercial detector, Wappalyzer.

4) Our experimental results show that PTDETECTOR can outperform LDC and Wappalyzer and can identify Webpack-bundled libraries.

II. Background and Motivation

A. JavaScript Library in Front-end

JavaScript libraries are commonly designed to adapt to different runtime environments. The library's APIs are composed of functions wrapped in objects. These objects are registered in the global context of the browser runtime, allowing the library's APIs to be globally available. Listing. 1 uses 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
6      // Define properties
7      lodash.chain = function(value) {
8          var result = lodash(value);
9          result.__wrapped__ = value;
10         return result;
11     }
12     lodash.differenceBy = ...
13     ...
14
15     // Export lodash
16     window._ = lodash;
17 }.call(this));
```

Listing 1. Simplified Lodash Browser Initialization Steps.

Listing. 1 presents a few key steps of the Lodash browser initialization, which is executed when the library is loaded. 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 7 - Line 13, various jQuery APIs (`chain`, `differencyBy`, and others) are registered as `lodash` object properties. Finally, in line 16, the `lodash` object is exposed to the identifier _ in the global context, i.e., registered as a property of `window`[3].

There is no trivial way to know what library runs on a web page. Since JavaScript libraries register objects to global scopes, analyzing property names during browser runtime is an important means of detecting front-end libraries. This approach's details will be expanded in Sec. II-B1.

B. Existing Detection Methods

Existing detection methods can be divided into two categories: dynamic and static. Dynamic methods detect JavaScript properties during browser runtime, while static methods apply matching on various web content, including cookies, DNS records, HTTP response headers, HTML source code, JavaScript source code, and more. In the following subsections, we elaborate on these detection methods based on one free tool (Sec. II-B1) and one commercial tool (Sec. II-B2).

1) Library-Detector-for-Chrome (LDC): LDC is the most popular (based on GitHub star) open-source JavaScript library detector. It was created in January 2010 and is still being updated today. It has 600+ stars on GitHub [5] and 10,000+ users on the Chrome Extension Store [6]. As a browser extension, LDC uses dynamic methods to detect libraries. Listing. 2 is a simplified JavaScript snippet of the LDC source code used to detect Lodash.

```
1 function testLodash () {
2     var _ = window._,
3         wrapper = _.chain(1);
4     if ( _ && wrapper.__wrapped__) {
5         return {version: _.VERSION || UNKNOWN};
6     }
7     return false;
8 }
```

Listing 2. Dynamic Detection method of Lodash in LDC.

Listing. 2 examines two JavaScript properties: _ and `_.chain`, in the global context. The Lodash function `chain` will return a `LodashWrapper` object containing the `__wrapped__` property (line 9 in Listing. 1). In line 3 of Listing. 2, LDC calls the `chain` function and assigns its return to `wrapper`. Next, in line 4, _ and `wrapper.__wrapped__` are checked. If both of them exist, Lodash is assumed to be present, and then, in line 5, the library version is retrieved from the property `_.VERSION` (return `UNKNOWN` if not found).

2) Wappalyzer: Wappalyzer[4] is one of the best commercial web technology profilers. Its browser extension has 2,000,000+ users on the Chrome web store [7]. In addition to libraries, it detects content management systems, e-commerce platforms, server software, and analytical tools. In this paper, we focus only on its ability to detect libraries. Wappalyzer published part of its detection details on GitHub [8]. Listing. 3 is a simplified JSON snippet of its meta-code that detects the Lodash library.

```
1 "Lodash": {
2     "js": {
3       "_.VERSION": "^(.+)$\\;version:\\1",
4       "_.differenceBy": ""
5     },
6     "scriptSrc": "lodash.*\\.js",
7 }
```

Listing 3. Meta Data in Wappalyzer for Lodash Detection.

Two patterns are given in Listing. 3, containing both dynamic and static methods. The "js" field, line 2, utilizes JavaScript properties to perform the detection. It examines two properties in the global context: `_.VERSION` and `_.differenceBy`. Line 6 provides a static method: the `scriptSrc` field is matched with URLs of JavaScript files included on the page. If any loaded JavaScript file name matches the regular expression, Lodash is assumed to be

[2]A modern JavaScript utility library delivering modularity and performance.

[3]Code running in a web page share single global object `window`.

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

detected. Compared with LDC, Wappalyzer's detection policy is more lenient — as soon as one of these three conditions is met, Wappalyzer assumes that Lodash is detected.

C. Limitation of Existing Methods

Existing detection methods inevitably lead to false detections. Due to the lack of namespaces in JavaScript, global property conflicts are common among JavaScript libraries; i.e., different libraries often define properties with the same name on the *window* object. Jibesh et al. [9] analyzed 951 libraries and found that one out of four is potentially conflicting. Therefore, detection based on a few properties can easily lead to false positives (Sec. II-C1). Moreover, the interference of packers such as Webpack(Sec. II-C2) and the labor cost (Sec. II-C3) required to maintain high-level detection are also major limitations.

1) False Positive: The selection of property is crucial to the accuracy of the dynamic detection method. On Netflix [5] home page, we found that LDC misidentified Underscore.js [6] library as Lodash. After manual inspection, this false identification can be attributed to two aspects. First, Underscore.js also defines a global _ property and a `_.chain` function, the same as Lodash. Second, we found that the website developer modified the Underscore.js library and defined the `__wrapped__` property in the return of `chain`, which satisfies the condition (Listing. 2 line 4) for mistakenly detecting Lodash in the LDC. For Wappalyzer, the situation is even worse. It determines that Lodash exists as soon as it detects the presence of `_.VERSION` ((Listing. 3 line 3), which is also present in the Underscore.js library. Therefore, Wappalyzer completely loses the ability to distinguish Lodash from Underscore.js.

The static detection method is not reliable either. It is a common practice for web developers to reorganize library files when importing locally. For example, on the "weather.com" web page, we found Wappalyzer misidentified Lodash because it matched a loaded local JavaScript file named "46202.lodash.0c9c71f173e278ac6235.js". This file name satisfies the regular expression given at line 6 in Listing. 3. But when we open this file, its contents are actually "react .production.min.js", a component of the React framework.

2) False Negative: False negatives are often seen due to the widespread usage of the Webpack, which is a module bundler published in 2012 to solve the global variable conflict problem. According to an industry report [10], as of March 2023, there are 15,448,980 websites built with it. Webpack wraps library APIs that would otherwise be defined globally into a local scope and splits a library file into multiple chunks to be loaded separately. As a result, libraries bundled by Webpack are no longer detectable by any existing tools.

3) Labor Cost: Another limitation of existing tools is that they require continuous human resources to maintain high accuracy. Over 4,000 front-end JavaScript libraries are currently on Cdnjs alone, each constantly being updated with newer versions. For example, React has 531 versions recorded on Cdnjs. Trying to adapt the detector to so many libraries and versions manually can be time-consuming and error-prone.

[5]A well-known TV streaming website: https://www.netflix.com/

[6]A library that provides functional programming helpers.

III. Design

Our tool PTDETECTOR follows the dynamic detection approach — using the libraries' runtime properties to identify. However, unlike LDC or Wappalyzer, which only validates a small number of properties, PTDETECTOR utilizes the property tree as the feature and detects the library by runtime-matching similar tree structures. We name such tree structure built on property relationship as *pTree*, elaborated in Sec. III-B. Fig. 1 shows part of Lodash's and Underscore.js's pTrees. The complete pTrees of these two have 258 and 157 vertices respectively. The rich property information allows PTDETECTOR to effectively distinguish even similar libraries, thus avoiding false positives mentioned in Sec. II-C1. Besides, PTDETECTOR is not limited to matching only in the global context - the search scope is the entire pTree rooted at the *window* of the page - so it can detect Webpack-bundled libraries.

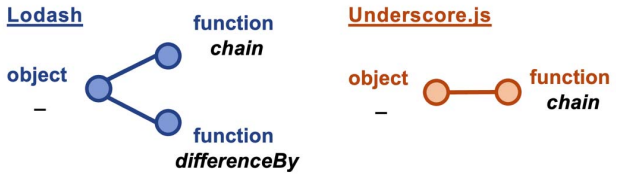

Fig. 1. Part of pTrees of Lodash and Underscore.js.

Most importantly, PTDETECTOR presents a way to automatically collect the pTree using a self-built web client. In this paper, we propose an algorithm to assign different weights to each vertex of the pTree, allowing the matching to accommodate small library changes. These changes may come from the library users' modifications or version differences. With the automation pipeline, a large number of libraries' detection features can be collected quickly, which gives PTDETECTOR a considerable advantage over existing tools regarding the number of libraries it can detect.

A. System Overview

PTDETECTOR is a browser extension to give scores to all possible JavaScript files loaded on the web page. A score of 100 indicates an exact match, and a score of 0 indicates no possibility of existence. Note that PTDETECTOR's detection target is the file, not the library. One library may contain multiple JavaScript files. The discussion of extending file detection to library detection is in Sec. IV-A1.

PTDETECTOR uses pTree as the detection feature. The definition and generation algorithm of pTree are given in Sec. III-B. Several processing steps on the pTree are necessary to promote detection effectiveness, including random-generated vertices removal to eliminate non-determinism; root-pruning to prevent the impact of dependencies; credit assignment, which allows matching to accommodate small changes; tree trim to save storage space; and inverted indexing to

increase matching efficiency. These steps are expanded in Sec. III-C with the workflow graph in Fig. 2. Finally, the pTree matching algorithm is applied during browser runtime detection, whose detail is disclosed in Sec. III-D.

B. Property Tree: pTree

1) pTree Definition: As the core concept of PTDETECTOR, pTree refers to a tree formed by the property relationship between JavaScript variables at a runtime frame. We propose this novel data structure to achieve a more accurate characterization of the objects registered by JavaScript libraries and to enable better runtime library detection based on tree similarity matching. Formally, if we use the symbol $[\![f]\!]$ to denote the set of all JavaScript variables at a runtime frame f, and ζ_v to denote the variable represented by the vertex v, the pTree T_f is a tree that meets following requirements:

For each vertex $v \in V[T_f]$, we have $\zeta_v \in [\![f]\!]$;
For each edge $(p, c) \in E[T_f]$, where p is c's parent,
we have ζ_p is an object, and ζ_c is a property of ζ_p.

For each vertex v, we assign it three attributes: ζ_v's name, type, and value (only if ζ_v has a meaningful value). In the later tree-matching algorithm, a vertex is considered matched only if all three attributes match. Vertex types' categories and corresponding values are shown in Table I. JavaScript value types can be divided into two categories: *primitive types* and *object types*. Primitive types include null/undefined, number, string, and Boolean. For null/undefined, it does not have a meaningful value. For numbers and strings, to save storage space, their values are shortened[7] as shown in Table I.

TABLE I
SIX VERTEX TYPES OF PTREE USED IN THIS PAPER.

Category	Type	Value	Children
primitive types	null / undefined	-	
	number	v.toFixed (3)	
	string	v.slice (0, 10)	
	Boolean	v	
object types	array / set / map	v.length	
	other object	-	✓

Any JavaScript value other than primitive types is an object. Here we separate object type into two classes: one is "array / map / set"; another is "other object". Array, map, and set are JavaScript built-in objects used to store elements (i.e., properties from the object's view). Therefore, we use the number of elements as their value and do not recursively check the properties of these vertices to avoid oversizing the tree. As a result, among the six vertex types in Table I, only the "other object" can have children vertices in the pTree we build.

2) pTree Generation: Given a variable v as root, we can generate a pTree by the following steps. Step I creates a vertex v and determines its type. Primitive types can be determined by the `typeof()` function; Object types can be determined by comparing objects' prototype[8]. Step II, based on v's type, gets v's value following the instruction in Table I. Step III, if n belongs to "other object", uses`Object.getOwnPeopertyNames(n)`[9] to get ζ_v's properties. For each of them, repeat Step I and append it as v's child. Notice that if a property of v points to an ancestor variable, then we should skip this property; otherwise, the algorithm will not end.

We leverage a trivial localhost client to generate the pTree. The client will host an empty web page to load the target JavaScript file. Once the file is loaded, we use *window* as the root for pTree generation. In this process, we only care about the new global variables defined by this file. On an empty web page, 850 properties are already defined on the *window* object in our browser environment. We omit these native properties. Subsequent mentions of generating pTree on the web page all follow the above process.

C. Feature Generation

Fig. 2 depicts PTDETECTOR's feature generation system workflow. The input to the system is a JavaScript file list. Each entry includes basic file information and its dependencies. PTDETECTOR will load the file with dependencies in localhost web pages and generate the pTree. The system's output is a JSON file "pt.json" containing all pTree information of these files. This workflow has three main sections. The first part is the generation of pTree while eliminating the impact of dependencies (Sec. III-C1). The second part is the post-processing of the pTree, whose core task is vertex credit assignment (Sec. III-C2). In the end, all generated pTree will be stored in an inverted indexing layout JSON file (Sec. III-C3).

1) Dependency Elimination: Some library files need other files to be loaded before them, which we call *outer dependencies*. For ease of use, some libraries copy the required dependencies directly into their own files, and we call them *inner dependencies*. The existence of dependencies in the feature generation stage can increase the possibility of false positives during detection. Assume the target file A has a dependency B. Because B must be loaded to run A, the generated feature will contain information from both files.

To remove the interference of dependencies, we use two clients to generate pTrees separately and calculate the difference between two pTrees. For the first localhost client (upper one in Fig. 2), we load outer dependencies and inner dependencies to get the dependency pTree dpT; For the second localhost client (lower one), we load the target file and its outer dependencies. In addition, we observed that some files define random variables[10]. To eliminate the randomness in detection, we generate pTree twice in the second localhost client and remove all different vertices to get a stable full pTree fpT. Then we remove those vertices in fpT that are also present in dpT. We call this process *root-pruning* and refer to the

[7] As far, our experiment using the shortened value has not led to ambiguity.

[8] For example, we can determine whether an object `O` is an array using the such equation: `Object.getPrototypeOf(O)===Array.prototype`

[9] `Object.keys(n)` can also return properties of an object, but non-enumerable ones are omitted. Some libraries may register all properties as non-enumerable. Velocity.js (v 2.0.6) is a case.

[10] Some libraries, such as jQuery, use random numbers to name some properties, which will result in a lower match score, thus reducing the recall.

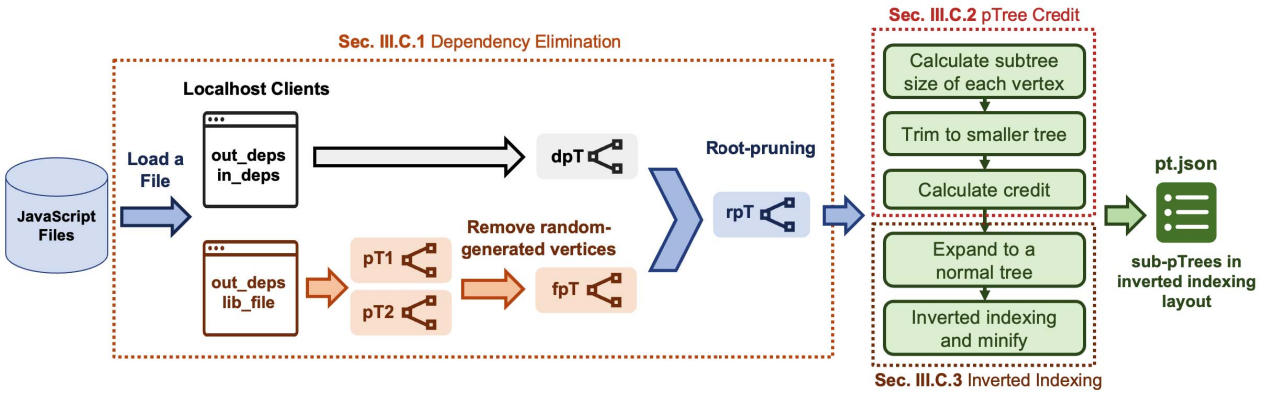

Fig. 2. PTDETECTOR Feature Generation Workflow.

tree after deletion as the root-pruned pTree rpT. Simply put, $rpT = fpT - dpT$.

Algorithm 1 Root-pruning Algorithm

Input: full pTree: fpT, dependency pTree: dpT
Output: root-pruned pTree: rpT

1: $rpT \leftarrow new$ Vertex$(window)$
2: $Q1, Q2$:= queue initialized with fpT's / dpT's root vertex
3: **while** $Q1 \neq \varnothing$ **do**
4: $u, v \leftarrow$ remove vertex from the front of Q1 / Q2
5: **for** each child s of u **do**
6: **if** $\exists\ t \in v$'s children, having $s = t$ **then**
7: insert s / t to the end of $Q1$ / $Q2$
8: **else**
9: append the subtree rooted at s as rpT's child
10: **end if**
11: **end for**
12: **end while**

Algo. 1 is the pseudocode of the root-pruning algorithm. The inputs to the algorithm are two pTrees: fpT and dpT. The output is the root-pruned pTree rpT. First, we create a new vertex, representing *window*, as rpT's root in line 1. Then, in line 2, two queues are initialized for a BFS from line 3 to line 12. While traversing children vertices in fpT, we check whether there is an identical vertex in dpT (line 6) — i.e., the path and three attributes of the vertex are all identical. If not, in line 9, we append the subtree rooted at this vertex as the child of rpT and stop traversing on this subtree.

Fig. 3 uses jQuery UI to demonstrate the root-pruning procedure. jQuery UI is a UI library built on top of jQuery. The left part of Fig. 3 shows the simplified pTree generated from the "jquery-ui.js" file. Those vertices also present in jQuery pTree are colored black, and others are colored orange. In Algo. 1, we detect all orange subtrees in fpT and combine them into a new tree rpT, shown in the right part of Fig. 3. Considering that rpT needs to be expanded into a complete tree for matching, each subtree should have its original path recorded. For example, in Fig. 3 rpT, the original path of the subtree rooted at the vertex labeled "slider" is "[jQuery, fn]".

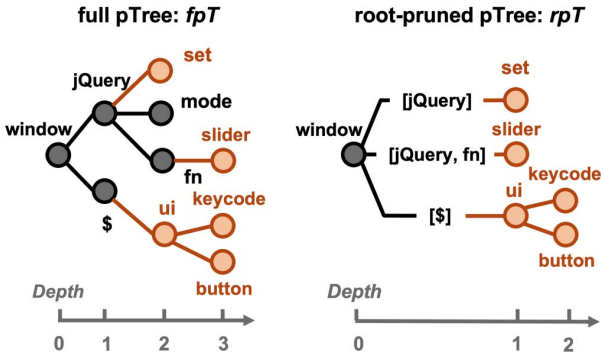

Fig. 3. Root-prune Demonstration (jQuery UI library).

2) pTree Credit: In pTree, each vertex is not equally important for prediction. In order to make the matching able to accommodate small changes from the library users' modifications or version differences, we assign each vertex a credit. The total credit of one pTree is set to 100. We design the vertex credit assignment rule based on two intuitions:

1) The smaller the depth, the higher the importance;
2) The larger the subtree size, the higher the importance.

The first intuition is that the high-level architecture of the library generally does not change. What changes are specific values or functions, i.e., the leaf vertices in a pTree. The second intuition comes from the following observation: the number of children of a vertex can reflect the number of APIs provided by the corresponding object, and objects with more APIs are more representative.

Now we define these intuitions formally. For a vertex v in the pTree, let d denote depth of v, D denote depth of pTree, $|T_v|$ denote the number of vertices of the subtree rooted at v, then the credit of v can be calculated using the equation:

$$credit = \frac{2^{D-d}}{2^D - 1} \cdot \frac{|T_v|}{\sum_{u.depth=d} |T_u|} \cdot 100 \qquad (1)$$

The first half of Eq.(1) ensures that the sum of credits in one layer of the pTree is twice that of the next layer. The credit of depth zero is 0, because *window* will not participate in

the matching. The maximum credit starts from depth one and decreases in geometric progression from layer to layer. The second half of the equation shows that vertices in one layer are assigned credits equally using subtree size as weight.

Another necessary operation is trimming the tree size. Some libraries have pTree with over 10,000 vertices. Using all the vertices as the feature will take up too much browser space and slow the detection speed. To retain high-credit potential vertices, trimming follows such preferences: (1) drop vertices with deeper depth; (2) at a given depth, drop vertices with smaller subtree size. Note that the trimming operation will lose subtree size information while discarding redundant vertices, which is a key parameter in Eq.(1). Hence, in the post-processing stage (green boxes in Fig. 2), the first step is to calculate the subtree size of each vertex in the rpT. Then, the pTree will be trimmed to a smaller tree based on vertex number and depth requirements. After that, the credit calculation will be conducted on the trimmed tree.

3) Inverted Indexing: To enable matching, the *rpT* will be expanded into a normal tree after credit assignment. Fig. 4 left is the expanded tree from the *rpT* in Fig. 3. Intermediate vertices (blue dotted nodes) added according to subtree path records will be assigned zero credit. Here the root vertex "window" is no longer needed because it is not a property defined by the library. We remove the root and split the pTree into sub-pTrees (right part in Fig. 4). The root vertex name of each sub-pTree is the *identifier*. In our example, file "jquery-ui.js" has two identifiers: "jQuery" and "$".

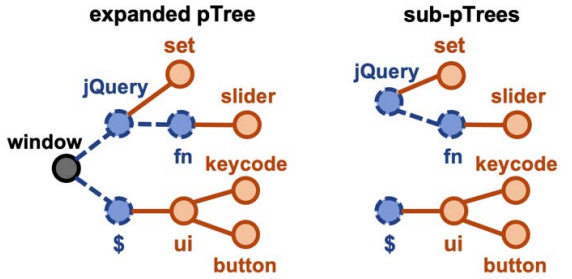

Fig. 4. Sub-pTree generation. Blue dotted vertices are added vertices.

To promote the efficiency of later tree matching, we organize generated sub-pTrees in the inverted indexing layout. Specifically, we combine all sub-pTrees with the same identifier and use it as their index. Fig. 5 shows an example of the inverted indexing table. The leftmost column lists all the identifiers, which map to sub-pTrees in different files. When one identifier is detected during runtime, its mapping sub-pTrees will be considered in the matching calculation. In the end, the inverted indexing table is stored in a JSON file "pt.json", and we minimize it into a compact representation.

D. Runtime Detection

We implement the PTDETECTOR detection ability as a browser extension. Once the extension's button is clicked, the detection procedure will start on the current web page. The

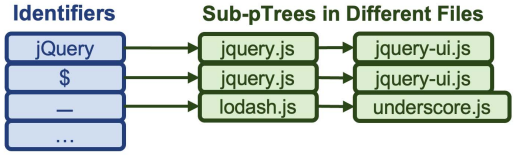

Fig. 5. Inverted indexing table mapping identifiers to sub-pTrees.

libraries' scores will be listed as the detection result in the extension popup window.

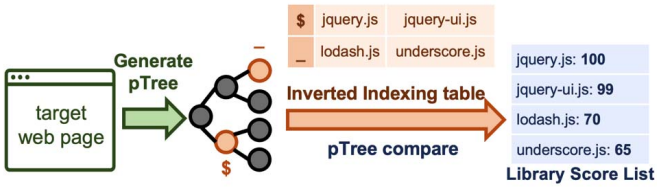

Fig. 6. PTDETECTOR Detection Workflow.

Fig. 6 depicts the detection workflow of PTDETECTOR. First, it generates the pTree rooted at *window* on the target web page. We limit the max depth of pTree to three to avoid the tree being too large. Then, we traverse the pTree and collect all vertices that have the same name with identifiers (e.g., "$" and "_" in Fig. 6). Next, based on the inverted indexing table, we use these vertices as targets for tree similarity computation (Algo. 2) with all relevant sub-pTrees indexed by the identifier. Each sub-pTree corresponds to one file, so after the computation, each file can get a score that represents the similarity of that file at the current vertex. There are two details to consider here. First, one file may link to multiple sub-pTrees. Hence, if several vertices with the same path index to a single file, the file score should be the sum of these vertex scores. Second, one web page may import a JavaScript file multiple times in different places. Thus, if one file is detected under more than one path in the web page pTree, we take the highest one as the final score of this file on the current page. Finally, we can get a list of all detected JavaScript files and display them in score descending order.

Algorithm 2 pTree Similarity Comparison Algorithm

Input: target pTree: tpT, sub-pTree: spT
Output: similarity score: S

Initialization: $S \leftarrow 0$
Match vertex count: $cnt \leftarrow 0$
for each vertex $u \in V[tpT]$ **do**
  for each vertex $v \in V[spT]$ **do**
    if $u.path = v.path$ **and** $u.name = v.name$ **and** $u.type = v.type$ **and** $u.value = v.value$ **then**
      $S \leftarrow S + v.credit$, $cnt \leftarrow cnt + 1$
    end if
  end for
end for
if $cnt = 1$ **then**
  $S \leftarrow 0$
end if

Algo. 2 shows the detail of pTree similarity comparison. There are two inputs: *tpT*, the pTree generated from a variable in the browser context, and *spT*, a sub-pTree from the inverted indexing table. The algorithm will traverse two pTrees and examine all match vertices. Two vertices match if and only if they have the same tree path and three attributes (name, type, and value) mentioned in Sec. III-B1. The value of the score is the sum of the credits of all matching vertices. Besides, we use a variable cnt (line 2) to record the matched vertex number. To reduce false positives, if only one vertex matches, we set the score as zero (line 11).

IV. Evaluation

This section investigates the version agnostic detection ability of PTDETECTOR on front-end JavaScript libraries on real-world web pages. In Sec. IV-A1, we discuss our dataset. We answer two research questions: RQ1 (Sec. IV-B1), how does PTDETECTOR compare to LDC and Wappalyzer? RQ2 (Sec. IV-B2), what are the best settings of our tool? Experiment data are also provided on GitHub [11].

A. Experiment Setup

1) Library Collection: To make the comparison unbiased, the target libraries are taken from the intersection of the technique lists[11] of LDC and Wappalyzer. As of March 15, 2023, 112 web techniques (including libraries) can be detected both by LDC and Wappalyzer, among which we need to filter out front-end JavaScript libraries. However, it is difficult to determine whether a technique belongs to the front-end JavaScript library since many techniques can accept multiple runtime environments. Considering that almost all front-end libraries mount their code on CDN for convenient HTML hyperlink importing, we pick all techniques that have code mounted on the Cdnjs platform or have CDN links provided on their official websites. Based on this, 83 techniques meet our requirements.

For each library, we use its latest version for feature generation. One library version may contain multiple JavaScript files, and we assume that the library is present if and only if at least one of its JavaScript files is loaded. Thus, adding all files of a library version into feature generation is unnecessary. In most cases, multiple files within a version have inter-dependencies. We call those that do not rely on other files as the library *base file*s. It is obvious that a library has at least one base file, and if the library is imported into the page, then at least one of its base files must exist. As a result, we can only consider library base files as the detection target. If a library has more than one base file detected on the web page, then the similarity score of the library is taken as the highest score among the base files. Most libraries provide two file versions for one snippet of JavaScript code — an original and a minified one. Since they do not differ under dynamic detection, we use only the minified version. This results in 99 JavaScript base files out of 83 libraries as our experiment dataset.

[11]Detectors normally disclose the web techniques they can detect on their official websites, including libraries.

Acquiring the accurate dependency information of each file is critical to detection precision. However, due to the lack of a unified management system, there is no trivial way to determine the dependencies of front-end libraries. For each base file, we manually check its inner and outer dependencies. Outer dependencies can be easily obtained from the library's official website instructions, while inner dependencies are hard to determine. Based on their source code, we carefully compare and verify other libraries contained in the file. In the end, we found 14 base files requiring outer dependencies and 8 base files having inner dependencies.

2) Website Collection: First, we select the top 200 websites from the SEMRUSH websites ranking [12], which is based on the US traffic on all categories of websites. We use the home page of each website as our testing web page. The ground truth of libraries these web pages use is necessary for the later comparison. Considering that no existing ground truth is provided, we determine ourselves using the following strategy. First, we manually apply three detectors — PTDETECTOR, LDC, and Wappalyzer — on each web page. To discover as many potential libraries as possible, here we set PTDETECTOR score threshold t as 0, i.e., library is assumed present as long as the score is larger than zero. For those pages where most of the content is displayed only after login, we use a temporary account for login and then perform library detection. Next, we take a union of all libraries detected by the three detectors[12], and carefully verify the existence of these libraries. Verification includes comparing web page JavaScript code, checking HTML information, investigating detectors' source code, and browsing the development blogs of the websites. During this process, we found that many web pages only contain jQuery or core-js, the most common two front-end libraries. To increase the diversity of libraries in the experiment, we exclude web pages that only contain jQuery or core-js and those that do not contain any library. Finally, with great effort, we arrive at a dataset that contains 80 web pages with 36 different libraries and 306 library occurrences.

3) Specification: We implement the detection component of PTDETECTOR as a Chrome extension. All the experiments are conducted on macOS Ventura (V 13.2.1) with an Apple M1 chip and 8G memory. All the web pages are opened on Chrome 110.0.5481.177 (Official Build) (arm64). This configuration is close to how everyday users use the browser.

B. RQ1: Comparison of Detection Tools.

1) Feature Analysis: We apply the PTDETECTOR feature generation workflow on our library dataset — 99 JavaScript files from 83 libraries. During pTree generation, we limit the max number of vertices to 10,000 and the max depth to 100, and get 99 pTrees in total. Table II lists size (number of vertices in the pTree), depth, number of back edges, and number of identifiers of each pTree. Back edges occur in

[12]Here, we assume that libraries not detected by any detector are not present. On the one hand, the probability of being missed by all three detectors simultaneously is low. On the other hand, these missed ones will not affect the comparison result between detectors.

the pTree generation algorithm. We removed them during the feature generation to prevent cycles. The number of identifiers equals the number of sub-pTrees for each file.

TABLE II
PTREE STATISTICS.

	Size	Depth	# Back Edges	# Identifiers
Average	1239.6	4.6	9.7	4.1
Median	112	4.0	0	1
Max	10000+	15	244	68
Min	3	1	0	1

Table II shows that the range of the pTree sizes is large. The library with the smallest pTree size, Web Font Loader[13], only provides one function and one object to wrap the function. While the library with the largest pTree size, Ink[14], has a pTree with over 10,000 vertices. The number of back edges ranges from 0 to 244, and 64 pTrees do not have a back edge. In most cases, back edges come from the copy of the reference to the *window* or the outermost wrapping object. The number of identifiers ranges from 1 to 68, and 57 files only have one identifier, which implies just one outermost object is used to wrap all contents.

The distributions of pTree's size and depth are shown in Fig. 7. Most sizes are between 10 and 1,000, and most depths are between 1 and 6. The good news is that only a few pTrees (9 / 99 = 9.1%) have less than ten vertices since too few vertices will affect the detection accuracy. On the other hand, too many vertices will reduce the detection efficiency, so pTree trimming is necessary. In this research question, we trim all pTrees with size and depth limits as 50 and 5.

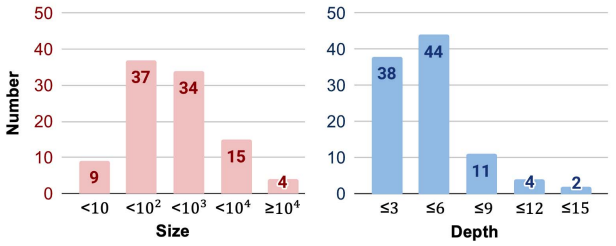

Fig. 7. Histogram of pTree's Size and Depth Distribution.

2) Web Page Detection: We load generated pTree features into PTDETECTOR and apply it to our dataset. After opening a web page, we wait 20s to ensure all libraries finish loading. PTDETECTOR traverses the first three layers in the web page's pTree to match identifiers and checks pTree similarity (Algo. 2). PTDETECTOR detects 289 library occurrences (score > 0). Among them, 263 have identifiers matched in the first layer, 19 in the second, and 7 in the third. Libraries with identifier layer depth larger than one are not imported in the browser's global context (may utilize techniques like packers) and thus cannot be found by other detectors. This means that our tool has the full ability to detect bundled libraries.

[13]https://github.com/typekit/webfontloader
[14]https://ink.sapo.pt/

Table III shows the detection result comparison of PTDETECTOR, LDC, and Wappalzer. We use three metrics to measure the detection performance — accuracy, precision, and recall[15]. Surprisingly, LDC as an open-source software exceeds the commercial tool Wappalyzer on all three metrics. Therefore, we use the better-performing LDC as the benchmark for PTDETECTOR. Fig. 8 presents the detection performance of PTDETECTOR under score threshold t ranging from 50 to 85. The dashed lines in the figure mark the precision and recall value of LDC as benchmarks.

TABLE III
DETECTION PERFORMANCE COMPARISON.

	LDC	Wappalyzer	PTdetector		
			$t = 57$	$t = 70$	$t = 77$
Accuracy	99.10%	98.63%	99.41%	99.43%	99.25%
Precision	**96.59%**	94.61%	**96.84%** ↑	100%	100%
Recall	**83.33%**	74.51%	90.20%	87.58%	**83.66%** ↑

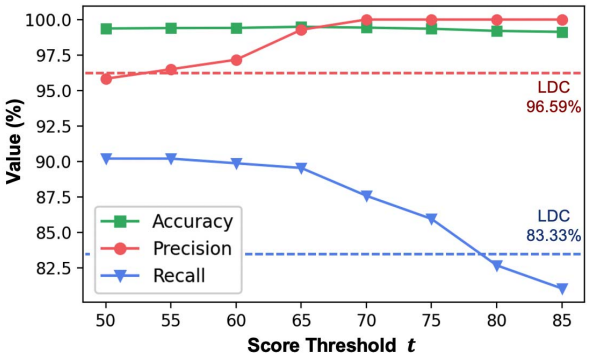

Fig. 8. PTDETECTOR Detection Performance (Compared with LDC).

In Fig. 8, the accuracy of PTDETECTOR always maintains a high level. This is because the value of TN far exceeds that of TP, FN, and FP. A website loads an average of 3.8 libraries in our ground truth dataset, but the tools check for 83 libraries. Therefore, most library tests are negative, resulting in large TN. As a result, accuracy can not exhibit performance differences in this experiment.

As threshold t increases, PTDETECTOR's precision keeps increasing while recall keeps decreasing. When $t \geq 57$, the precision of PTDETECTOR exceeds LDC's; when $t \leq 77$, the recall of PTDETECTOR exceeds LDC's ("↑" marks in Table III). Overall, when $57 \leq t \leq 77$, PTDETECTOR outperforms LDC and Wappalyzer on all three metrics. And when $t = 70$, PTDETECTOR achieves high precision and recall — 100% and 87.58%.

For each library, we record the number of its occurrences in our dataset and calculate the average score given by PTDETECTOR (0 if not detected). Table IV lists the top ten and last ten libraries based on avg. score ranking. Interestingly, the six libraries with an avg. score below 80 except Modernizr[16] all belong to frameworks. Strictly speaking, the

[15]Accuracy=$\frac{TP+TN}{TP+TN+FP+FN}$, precision=$\frac{TP}{TP+FP}$, recall=$\frac{TP}{TP+FN}$.
[16]A feature detection library mainly used on the back-end. The files mounted on Cdnjs are old versions, leading to a low average score.

JavaScript libraries and frameworks are different. The former is reusable code with a single primary use case. The latter is a set of JavaScript codes that provide pre-written code for everyday programming tasks to web developers. Most of the core code of the framework is on the back end, aiding the developer's work. The code mounted on CDN is commonly their runtime debugging tool, which is optional to load. As a result, PTDETECTOR failed to achieve satisfactory detection on these frameworks.

TABLE IV
PTDETECTOR DETECTION SCORE RANKING OF LIBRARIES.

No.	Library	avg. score	cnt	No.	Library	avg. score	cnt
1	Lo-dash	100	21	27	SWFObject	88.1	1
2	IfVisible.js	100	4	28	Backbone	87.8	8
3	WebFont	100	1	29	Knockout	82.1	1
4	Head JS	100	1	30	Bootstrap	79.5	7
5	Kendo UI	99.9	1	31	Ext JS	75.4	1
6	Prototype	99.8	1	32	Preact	74.7	1
7	Pusher	99.4	2	33	Modernizr	65	12
8	jQuery	99.1	60	34	React	11.4	28
9	RequireJS	99	12	35	Vue	0	3
10	Moment.js	98.7	6	36	Angular	0	1

Frameworks have various architectures, and automating their detection is hard. Luckily, the number of commonly used frameworks is small, and some provide official browser extension tools for detection: "React Developer Tools" for "React" and "Vue.js devtools" for "Vue". Hence, PTDETECTOR could combine these existing tools for better framework detection.

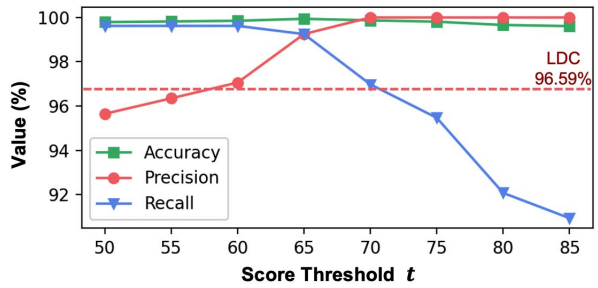

Fig. 9. PTDETECTOR Performance after Excluding Frameworks.

Fig. 9 shows the performance of PTDETECTOR after excluding six frameworks. Recall has been significantly improved compared to Fig. 8. When t reaches 68, the precision achieves 100%. At the same time, the recall is 98. 11%, much higher than the recall of 83.33% of LDC (out of range in Fig. 9).

RQ1 Conclusion: When the threshold is set within a reasonable range (57 to 77), PTDETECTOR's detection ability outperforms LDC and Wappalyzer in all metrics, even in the presence of libraries wrapped by packers.

C. RQ2: Best Trim Settings of PTDETECTOR.

In PTDETECTOR, we trim a pTree into a smaller version to prevent oversizing. The trim size and depth setting will affect the detection performance and efficiency. To answer this research question, we examine the performance, overhead, and space of PTDETECTOR under different trim settings. We fix the trim depth limit to five. According to Eq.(1), where vertex credit decreases in geometric progression as depth grows, the sum of credits outside of the first five layers is less than $1/32$, which has little effect on the final score. As a result, we only need to focus on investigating the impact of the size limit.

We apply PTDETECTOR with eight different size limits and repeat the detection steps in RQ1. We calculate three values — AUC, "avg. Time", and "Space per Lib", shown in Table V, to measure the detection performance, overhead, and space requirement of PTDETECTOR. AUC represents the area under the ROC curve, providing an aggregate performance measure across all score thresholds. "avg. Time" is the average time spent on detection for all web pages, starting from detection and ending at the result display. For each web page, we ran the test program five times and averaged the test time for this page to mitigate the impact of network fluctuations on timing. We use "avg. Time" to measure the overhead of PTDETECTOR. "Space per Lib" is obtained by dividing the size of the generated JSON file, which stores the pTree inverted indexing table, by the number of libraries.

TABLE V
AUC, TIME, AND SPACE UNDER EIGHT SIZE LIMIT.

Size Limit	AUC	avg. Time (ms)	Space per Lib (KB)
5	0.8688	790.16	0.40
10	0.8737	755.31	0.84
25	0.9342	777.69	2.02
50	0.9422	772.81	3.60
100	0.9374	806.77	6.36
200	0.9325	790.34	10.41
500	0.9241	770.56	18.84
1000	0.9192	777.20	28.35

Fig. 10 presents the data in Table V as line graphs. The figure shows that the AUC is low when the size limit is less than 10. The AUC rises sharply when the size limit increases from 10 to 50. However, after 50, the AUC decreases slowly until the size limit reaches 1000. This result is counter-intuitive at first glance, considering that a larger size limit implies richer information. We use the latest version of each library during feature generation, but the actual web page may use a different version. Thus, the larger the size of pTree, the more negatively it affects the detection performance when the version of the library from which pTree was built differs from the version of the library loaded by the web page.

The mid plot in Fig. 10 shows the "avg. Time" trend. We can see that the detection time does not show regular changes as the size limit increases. This occurs because the detection time is greatly affected by network fluctuations. Additionally, the detection time depends on the pTree structure of the web page and not the size of pTrees collected from libraries.

The plot on the right in Fig. 10 shows the usage of space. As the size limit goes up, "Space per Lib" shows approximately linear growth. If we consider using 50 as the size limit to store pTree information for all libraries on Cdnjs, the required

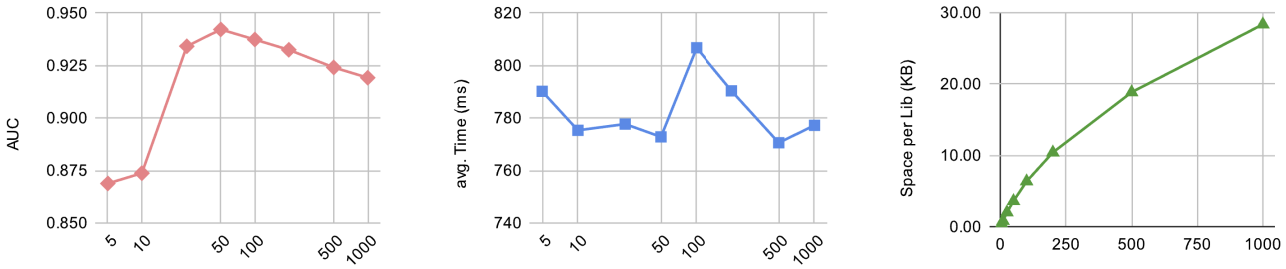

Fig. 10. The Trend of AUC (left), Time (middle), and Space (right) under Different Size Limits.

space would be $4366 \times 3.6KB = 15.31MB$. This is within the acceptable range for a browser extension.

RQ2 Conclusion: PTDETECTOR achieves the highest detection performance around a size limit of 50 with an average detection in less than 800ms. Space usage is acceptable (4366 libraries in less than 16MB).

V. Threats to Validity

Our analysis is limited to the top 200 trafficked websites, which are mainly large portal sites. Considering that the libraries that different types of websites use may be different that those in our data set, we cannot ensure that the experimental results of this paper generalize to all websites. Additionally, only the 36 most commonly used libraries are included in our experiments. For libraries not present in our data set, there may exist potential factors affecting the performance of the PTDETECTOR.

VI. Limitation

Although PTDETECTOR shows good performance, some limitations still exist. The first is the inability to detect library versions. Today, there are 2,509,859 library versions on Cdnjs. With a size limit of 50, over 8G space is needed to store all their feature information. One approach is to conduct two-layer detection, i.e., detecting the libraries first and then performing specific version detection after generating just-in-time pTrees for each version. This is our future work.

Another limitation lies in module detection. ES6 module is a new browser library importing mechanism that allows partial library loading. Our pTree matching algorithm is based on the assumption that the library is always loaded entirely. However, our observation shows that the ES6 module is not popularized among websites. Developers still prefer to use the traditional library loading method. None of the top 200 websites use the ES6 module.

The dependency requirement is also a limitation. PTDETECTOR requires accurate dependency information for each JavaScript file as input. However, dependency information can only be collected manually, greatly limiting the tool's automation level. In fact, PTDETECTOR, with a few modifications, is also capable of automatic dependency detection. This also is our future work.

VII. Related Work

Library Detection. Although, to the best of our knowledge, we are the first to investigate library detection for web applications, 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) [13] is a representative binary matching method that utilizes constant values as the detection feature and applies a frequency-based ranking method to identify the presence of libraries. OSSPolice [14] introduces a hierarchical indexing scheme to better use the constant information and the sources' directory tree. This data structure inspired us to design an inverted indexing table to store pTrees. Then BCFinder [15] makes the indexing lightweight and the detection platform-independent. B2SFinder [16] synthesizes both constant and control-flow features from binary based on their importance-weighting methods, giving more reliable library detection results. Xian Zhan et al. [17] conducted the first empirical study on existing Android library detection techniques and proposed tool selection suggestions. ModX [18] introduces a novel algorithm to detect partially loaded libraries via semantic module matching. Unfortunately, these methods cannot adapt to the web environment due to the vast differences between web and traditional applications.

Web Library Analysis. Many kinds of library analysis work have been done. Feldthaus et al. [19] present a pragmatic approach to check the correctness of TypeScript files with respect to JavaScript library implementations. Erik et al. [20] explore the concept of a reasonably-most general client and introduce a new static analysis tool for TypeScript verification. Patra et al. [9] present an automated method to detect JavaScript libraries' conflicts and show that one out of four libraries is potentially conflicting. Moller et al. [21] develop the tool Tapir that finds the relevant locations in the client code to help clients adapt their code to the breaking changes. Wyss et al. [22] propose a tool to programmatically detect hidden clones in npm and match them to their source packages. Their tool utilizes a directory tree as a detection feature, which does not apply to the front-end library.

Analysis of JavaScript. The JavaScript analysis research can be divided into two main topics: static analysis and dynamic analysis. According to an empirical study [23], static has been the most dominant research topic for client-side

JavaScript applications. Numbers of work focus on extending analysis scope, including dynamically loaded code [24], [25], dynamic features [26], [27], and DOM [28], [29]. Another static analysis trend is to improve the analysis precision by handling dynamic features and loops more elaborately. To address dynamic features, various hybrid approaches are proposed [30], [31], [32], [33]. For loops, researchers also proposed techniques to analyze them precisely [34], [29]. Because of the extremely dynamic nature of JavaScript, dynamic analysis is also an active research topic. Several testing techniques have been proposed to address language-specific features [35], [36], [37], [38]. To enhance the dynamic analysis coverage, crawling and dynamic symbolic execution techniques have been proposed [39], [40], [41].

VIII. Conclusion

JavaScript front-end library detection is a long-standing challenge. This paper introduces PTDETECTOR, which provides the first automated method to detect JavaScript front-end libraries on web pages. Using JavaScript library files and their dependencies as input, the system generates pTrees as the detection feature. Our experiments on real-world web pages show that PTDETECTOR can identify packer-bundled libraries and its detection results outperform LDC and Wappalyzer in all metrics when the threshold is reasonably set.

References

[1] S. Thorogood, "Es6 modules in chrome m61+," 2015, https://medium.com/dev-channel/es6-modules-in-chrome-canary-m60-ba588dfb8ab7.

[2] W3Techs, "Usage statistics of javascript libraries for websites," 2023, https://w3techs.com/technologies/overview/javascript_library.

[3] 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/.

[4] X. Liu, "Ptdetector on github," 2023, https://github.com/aaronxyliu/PTdetector.

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

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

[7] C. W. Store, "Wappalyzer chrome extension," 2023, https://chrome.google.com/webstore/detail/wappalyzer-technology-pro/gppongmhjkpfnbhagpmjfkannfbllamg.

[8] GitHub, "wappalyzer/wappalyzer," 2023, https://github.com/wappalyzer/wappalyzer.

[9] 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.

[10] BuiltWith, "Websites using webpack," 2023, https://trends.builtwith.com/websitelist/Webpack.

[11] X. Liu, "The experiment data of ptdetector on github," 2023, https://github.com/aaronxyliu/PTdetector-Data.

[12] SEMRUSH, "Semrush top websites ranking," 2023, https://www.semrush.com/website/top/global/all/.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] C. Yang, Z. Xu, H. Chen, Y. Liu, X. Gong, and B. Liu, "Modx: binary level partially imported third-party library detection via program modularization and semantic matching," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1393–1405.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner, "Staged information flow for javascript," in *Proceedings of the 30th ACM SIGPLAN conference on programming language design and implementation*, 2009, pp. 50–62.

[25] S. Guarnieri and B. Livshits, "Gulfstream: Staged static analysis for streaming javascript applications." *WebApps*, vol. 10, pp. 6–6, 2010.

[26] C. Park, H. Lee, and S. Ryu, "All about the with statement in javascript: Removing with statements in javascript applications," *ACM SIGPLAN Notices*, vol. 49, no. 2, pp. 73–84, 2013.

[27] C. S. Jensen, A. Møller, V. Raychev, D. Dimitrov, and M. Vechev, "Stateless model checking of event-driven applications," *ACM SIGPLAN Notices*, vol. 50, no. 10, pp. 57–73, 2015.

[28] S. H. Jensen, M. Madsen, and A. Møller, "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*, 2011, pp. 59–69.

[29] C. Park and S. Ryu, "Scalable and precise static analysis of javascript applications via loop-sensitivity," in *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[30] S. Guarnieri and V. B. Livshits, "Gatekeeper: Mostly static enforcement of security and reliability policies for javascript code." in *USENIX Security Symposium*, vol. 10, 2009, pp. 78–85.

[31] S. Just, A. Cleary, B. Shirley, and C. Hammer, "Information flow analysis for javascript," in *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients*, 2011, pp. 9–18.

[32] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Dynamic determinacy analysis," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 165–174, 2013.

[33] S. Wei and B. G. Ryder, "Practical blended taint analysis for javascript," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 336–346.

[34] E. Andreasen and A. Møller, "Determinacy in static analysis for jquery," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, 2014, pp. 17–31.

[35] A. Mesbah and A. Van Deursen, "Invariant-based automatic testing of ajax user interfaces," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 210–220.

[36] K. Pattabiraman and B. Zorn, "Dodom: Leveraging dom invariants for web 2.0 application robustness testing," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE, 2010, pp. 191–200.

[37] M. Mirzaaghaei and A. Mesbah, "Dom-based test adequacy criteria for web applications," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 71–81.

[38] A. M. Fard, A. Mesbah, and E. Wohlstadter, "Generating fixtures for javascript unit testing (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 190–200.

[39] A. Mesbah, E. Bozdag, and A. Van Deursen, "Crawling ajax by inferring user interface state changes," in *2008 eighth international conference on web engineering*. IEEE, 2008, pp. 122–134.

[40] A. Mesbah, A. Van Deursen, and S. Lenselink, "Crawling ajax-based web applications through dynamic analysis of user interface state changes," *ACM Transactions on the Web (TWEB)*, vol. 6, no. 1, pp. 1–30, 2012.

[41] M. Schur, A. Roth, and A. Zeller, "Mining behavior models from enterprise web applications," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 422–432.