JavaScript Library Version Detection

Vilim Pagon*, Bruno Skendrović†, Ivan Kovačević‡, Stjepan Groš§
University of Zagreb Faculty of Electrical Engineering and Computing
Zagreb, Croatia

***Abstract*—There are more than 1.6 billion websites today, and almost every one of them uses JavaScript libraries. Knowing that, it's very important to show problems that occur as a result of not paying enough attention to security, such as using outdated versions of JavaScript libraries, insecure libraries, and so on. This paper proposes an algorithm for JavaScript library version detection. The algorithm detects version of JavaScript libraries based on differences between neighboring library versions. It's designed in such a way that it can be run periodically and automatically on a server. The paper also presents results and efficiency of the algorithm on a smaller set of data collected from the Croatian Web space. The success of the algorithm in detecting the correct version is about 50%, and the range of probable versions is an additional 25%. From these results, i.e. the detected versions, we found that the JavaScript libraries used on the websites of the Croatian web space are not regularly updated. Limitations and also possible potential improvements to the algorithm are listed at the end of the paper.**

Keywords*—*JavaScript, version detection, cybersecurity, Croatian web space

I. Introduction

Due to the simplicity of using the Internet, the concept of security has become an essential part of the development of software products that use the Internet as an access and management medium. Vulnerabilities can occur at all stages of the creation process of a software product. In most cases, they are introduced unintentionally because they are not given sufficient attention during development. It is necessary to know what errors can occur in order to prevent this problem. Likewise, it is useful in this area to know how much damage bugs can do if they are abused, what potential threats could exploit them, and how to protect against said threats. An introduced vulnerability must be eliminated as soon as it is discovered so that it cannot be abused.

Security patches are one method of eliminating such vulnerabilities. They are used to update systems, applications or software by inserting code to close or "patch" security vulnerabilities. It is necessary to regularly update the JavaScript libraries used by the website to reduce their vulnerability and thus ensure the website's security. The mentioned practice should be standard for every developer, but it turns out that this is not the case [1]. The article shows how the number of requests for older versions of the library decreases very little, if at all, after the new version is released. Another part of the problem occurs when a patch is released, but the developers do not update the existing one, so the vulnerability in the code persists.

The aim of this paper is to present the developed technique for detecting the vulnerability of the website, i.e. the version of JavaScript libraries used by the website.

The paper starts with an introduction to the topic of the paper, and then the next chapter titled Related Work contains other papers and projects similar to this one. In the third chapter, the main terms and techniques used in the paper are introduced to facilitate the understanding of the text. After that, the idea of the developed detection method and its implementation are described in detail in the chapters IV. and V. The Experiments chapter presents the results of this detection method. The problems and limitations of the implementation are presented in the chapter VII. The last chapter is dedicated to final conclusions and possible improvements in future papers.

II. Related Work

Numerous scientific papers address the issue of static vulnerability detection. Most papers use variations of machine learning, deep networks, or a combination of static and dynamic analysis for detection. Some examples of scientific papers on this topic are Wang et al. [2] and Russel et al. [3]. Unlike the aforementioned scientific papers that focus on source code analysis of various programming languages, the analysis in this paper is based on JavaScript code. Papers that use vulnerability detection methods in JavaScript are Tripp et al. [4], Guarnieri et al. [5] and Kluban et al. [6]. This paper describes vulnerability detection in the JavaScript library used based on libraries with known vulnerabilities. Specifically, the developed tool detects the version of the library and determines the presence of one or more vulnerabilities based on some known vulnerabilities in JavaScript libraries. This method is performed based on the differences between neighboring versions of the original libraries downloaded from known CDN servers. Since this is a new method for detecting vulnerabilities in JavaScript libraries, it is difficult to compare it to previous methods, as after an extensive search of published papers, not a single paper was found that performs detection using differences between neighboring versions. Vulnerability detection usually uses some form of artificial intelligence, machine learning, or dynamic analysis, which is not the case in this paper. However, the obtained results, which will be shown later, can be seen as successful and show the whole method as potentially successful. There are also Chrome extensions that analyze the JavaScript libraries used on the current website, such as Wappalyzer [7]. The JavaScript library vulnerability repository used in this paper is the publicly available Retire.js [8]. This is a project that has developed a command line scanner that can be used to determine dependencies with known vulnerabilities in a web application.

III. Background

The detection method described in this paper is based on static analysis. Static source code analysis is a method for testing the source code of an application without executing the code [9]. Static analysis involves observing how the system, i.e., the solution, works.

The first two steps of JavaScript library version detection use hash values. The library hash value is a value calculated based on the source code of that library and passed to the function that calculates the cryptographic hash value. The value returned by this function is unique for each string. One of the functions used is the cryptographic hash function SHA-256 [10] which returns a unique value for each string.

The algorithm receives JavaScript code as input. It is the source code of a JavaScript library. At the beginning, the library and the version of the library were unknown, so this library is called an *unknown library*.

In contrast, *known libraries* are original JavaScript libraries downloaded from known CDN servers and used as a reference for a specific library, i.e. its source code.

Modified libraries are libraries from which redundant or non-functional parts of the code have been removed. These may be blank lines, spaces, new lines, comments, etc.

The third step uses *identifiers* to identify the library. Library identifiers are any keywords, variable names, function names, etc. that are characteristic of a particular library and do not appear in any other library. They are used to identify which library it is. They are selected from known versions of JavaScript libraries by analyzing their source codes and comparing them with each other to determine the identifiers that are unique to each library.

When detecting the JavaScript library version, the presence of a large number of minified libraries should be noted [11]. They pose a problem for detection because the code appears only in several and often only in one line. To solve this problem, reduction to a common form (caconic form), i.e., deminification, is used.

The reduction to canonical form - deminification - was performed using the Python library *JSBeautifier* [12]. Since minification detection was not performed, each library was reduced to a common form, which solved the problem of minimal differences between libraries, such as blank lines or spaces. For the rest of the program, it was assumed that all libraries were formatted the same and ready for comparison.

IV. Method of Version Detection

This paper is focused on static detection based on the source code of an unknown JavaScript library. The detection algorithm tries to find out which library and version that source code belongs to. The tool receives a JavaScript code as input for which the library version needs to be determined. The output, i.e. the detection result, is 10 versions of the library with the highest percentage of detection. Once the library version has been successfully determined, vulnerabilities can be searched in a publicly available repository with a list of vulnerabilities. It is a *Retire.js* [8] repository which contains a list of vulnerabilities for 26 of the most popular JavaScript libraries. Vulnerabilities are described as parameters *atOrAbove* and *below* that show the version in which the vulnerability was discovered, that is, which is the last version where this vulnerability is present. Detection was developed in steps where each step is performed separately and in a predetermined order. This paper discusses the fourth step of detection, which requires the first three steps to be completed for successful operation. These steps are shown in Figure 1.

1) The first step of detection uses a comparison of the hash value of each unknown library with the hash values of known versions of the libraries in the local database. The Library version is found if the hash values match, otherwise continue with the second step.
2) The comparison is done over modified libraries. The libraries have been modified in such a way that the removed lines of code do not affect the functionality of the rest of the code. Some of them are comments, empty lines, etc. The hash values of those modified libraries are recalculated and the first step procedure is repeated.
3) The third step is very important for detection. In this step, library name detection is done based on the identifiers. Identifiers are all keywords, variable names, function names, etc., which are characteristic of a certain library and do not appear in any other library. Each library has its own list of identifiers that are unique to it. The unknown library is compared to that list of identifiers and the results show which library it is with a certain percentage that indicates certainty. The obtained results are stored in the local database and used in the next step.
4) Previously obtained results are used in this step. For detection in the fourth step, a database of known versions of JavaScript library source codes was used. This data was collected from a well-known CDN server called cdnpkg [13] and saved in the database. Before detection itself, it is necessary for database of known JavaScript libraries to be supplemented with the differences between their neighboring versions. The differences are recorded locally in the *old* and *new* lists in the database. The list named *old* contains lines that were completely removed in the newer version, while the list *new* indicates the lines that were changed or completely added compared to the version before. These differences are obtained using the Python library difflib [14], which returns the

lines that are different between the two versions. In order to detect the version faster, a combination of the detected library name and the difference between neighboring versions of that library was used.

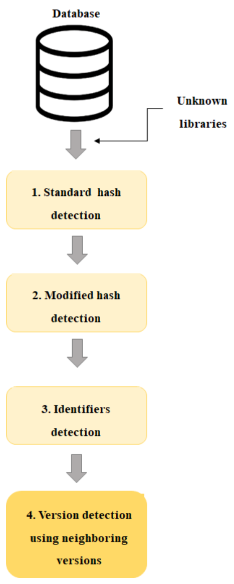

Database
Unknown libraries
1. Standard hash detection
2. Modified hash detection
3. Identifiers detection
4. Version detection using neighboring versions

Fig. 1: Steps of detection

Each subsequent detection step is more detailed and determines the version of the unknown library more accurately. Detection with hashes gives the most accurate results but gives the least number of detections, while the fourth step is less accurate but gives much more results with the most accurate library version information.

A more detailed description of the fourth step is described in the next chapter. Previously detected library's name was compared with the new lists of each version of the library. The hit percentage is determined by the ratio of the number of found lines to the total number of lines in the *new* list. The detection results are also recorded in the local database with other results. Based on these results, it is possible to further continue with a more detailed detection.

V. Developing a system for detection

A Python-based tool that automates the process was developed in order to detect the version of an unknown JavaScript library. The tool uses two MongoDB databases described in the next paragraph.

Two MongoDB databases were used in the detection process as can be seen in Figure 2. The first database contains the data on which the detection is carried out. This data was collected using a crawler in the laboratory and contains data from the websites of the Croatian Web space including HTML pages, JavaScript code used on those pages and some additional data [15]. In the following text, it will be called *Database with crawled data*. Second, the database was used to store results and auxiliary data in the process itself called *Results database*. Both databases are located on the private server where the detection is performed.

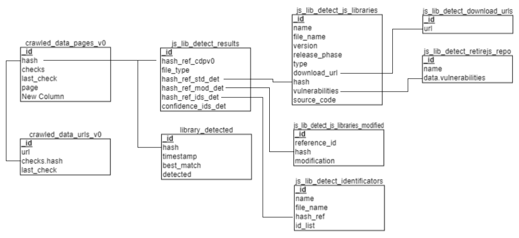


Fig. 2: Simplified structure of used databases

The first listed database, the *database with crawled data*, primarily serves as a database of collected data on which detection is performed, while the second one, *results database*, contains the data necessary for all steps of detection of an unknown library version. It also serves to store the results of individual detection steps for the purpose of further detection. The *database with crawled data* contains two essential collections. The first is a collection providing the source code that needs to be analyzed and the second is linked to the first in order to show which page it is about when the code is analyzed. That is shown in Figure 3.

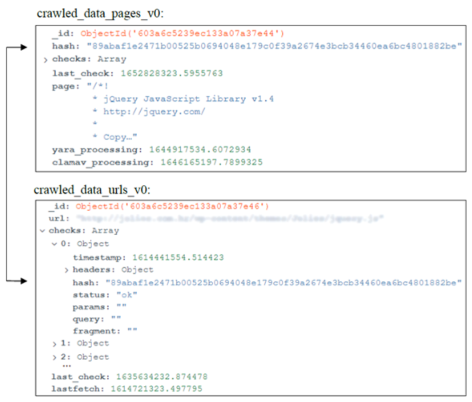
In the *results database* there is a collection containing the results of all detections:

- standard detection with hash values
- modified detection with hash values
- detection by identifiers

A separate collection was created for the results of detections. It was made due to the different structure of the results.

There are also some auxiliary collections:

- URLs of known library versions
- codes of known versions of JavaScript libraries
- codes of known versions of JavaScript libraries - modified
- a copy of the Retire.js repository - vulnerabilities
- identificators


crawled_data_pages_v0:

```
 _id: ObjectId('603a6c5239ec133a07a37e44')
 hash: "89abaf1e2471b00525b0694048e179c0f39a2674e3bcb34460ea6bc4801882be"
› checks: Array
 last_check: 1652828323.5955763
 page: "/*!
        * jQuery JavaScript Library v1.4
        * http://jquery.com/
        *
        * Copy…"
 yara_processing: 1644917534.6072934
 clamav_processing: 1646165197.7899325
```

crawled_data_urls_v0:

```
 _id: ObjectId('603a6c5239ec133a07a37e46')
 url: [redacted]
⌄ checks: Array
  ⌄ 0: Object
      timestamp: 1614441554.514423
    › headers: Object
      hash: "89abaf1e2471b00525b0694048e179c0f39a2674e3bcb34460ea6bc4801882be"
      status: "ok"
      params: ""
      query: ""
      fragment: ""
  › 1: Object
  › 2: Object
    ...
 last_check: 1635634232.874478
 lastfetch: 1614721323.497795
```

Fig. 3: Collections in the database with crawled data

Auxiliary collections show importance in various detection situations. Such as URLs of known library versions, so that subsequently the relevance of downloaded library versions is determined. Also, for detection by identifiers to be possible, it is necessary for those identifiers that are characteristic of a particular library to be stored somewhere. That collection contains lists of identifiers for each library and a hash value that represents the reference of the library to which the identifier detection result refers.

Each detection starts with a collection of known library versions and their source code. This collection is required at every step of discovery. It contains basic information needed for any detection, such as name, file name, version, release stage, type, hash, vulnerabilities, source code. The linked collection used in modified hash detection with hashes of known libraries is a collection containing the hashes of modified libraries.

To be able to continue with other more detailed detections, it's necessary to store the intermediate detection results in the database. All the results are in the aggregated collection of the results of all detections, which is linked to other collections as shown in Figure 2. It contains the results of standard and modified detection and detection using identifiers. The results are ordered according to hashes that represent references to the detected objects.

The created Python tool accesses the server's resources and uses the databases located on it at each detection step, which structure is shown in Figure 4.

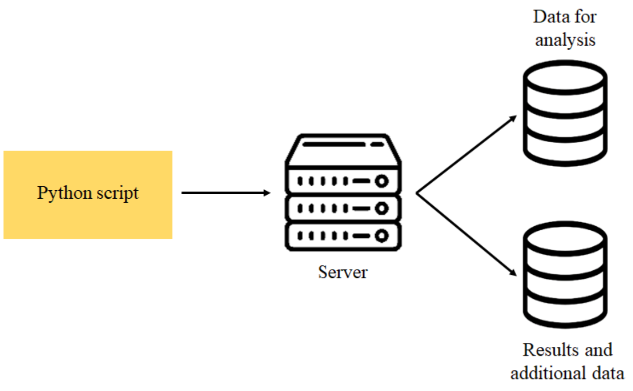
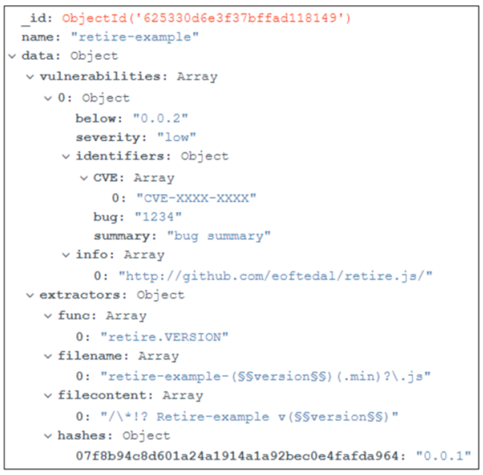


Fig. 4: System architecture

The version detection process starts by retrieving all JavaScript code from the collection with the results of previous detections. Filtering continues on the obtained records so that those libraries that were not detected by standard and modified detection using hash values remain in the process. Among the fetched records, those records detected using identifiers are added to the list of libraries to be analyzed. Furthermore, for each element from the previously created list, the source code is taken from the *database with crawled data* to which the record from the result collection refers. Each fetched source code of the fetched library is reduced to the canonical form using the tool Beautifier as well as the known libraries against which it is compared. Also, the library detected by the identifiers is retrieved to determine the name of the library with whose versions the algorithm must compare the unknown library. Once a library's name is determined, retrieve all versions of that library. And finally, the comparison process starts. Since the library's name is now known, it is necessary to determine the specific version. Lines of an unknown library reduced to canonical form are compared with a list of lines that did not exist in previous versions of each version of that library. The percentage of rows found determines the probability of a successful version hit. Based on match percentage, the top ten hits and the best hit are returned. After the comparison is complete, the results are saved in a separate collection created just for those results. Duplication is avoided by updating the existing document if it already exists in the library.

Detection results based on differences between neighboring library versions are saved in a separate collection. The reason for this is that the documents have a different structure than the previously described collection. This collection contains a hash field that points to the unknown library that was analyzed, a field that shows the most likely version of the unknown library, and a list of the top 10 possible versions. If the algorithm does not provide detection with 100% certainty, a more detailed analysis continues over the mentioned list of the next 10 most likely versions. If the library version is found, i.e. the algorithm is library version safe, using a collection that is a copy of the Retire.js repository, it is possible to determine if an already known library is vulnerable. A sample document from that collection can be seen in Figure 5.



```
  _id: ObjectId('625330d6e3f37bffad118149')
  name: "retire-example"
v data: Object
  v vulnerabilities: Array
    v 0: Object
        below: "0.0.2"
        severity: "low"
      v identifiers: Object
        v CVE: Array
            0: "CVE-XXXX-XXXX"
          bug: "1234"
          summary: "bug summary"
      v info: Array
          0: "http://github.com/eoftedal/retire.js/"
  v extractors: Object
    v func: Array
        0: "retire.VERSION"
    v filename: Array
        0: "retire-example-(§§version§§)(.min)?\.js"
    v filecontent: Array
        0: "/\*!? Retire-example v(§§version§§)"
    v hashes: Object
        07f8b94c8d601a24a1914a1a92bec0e4fafda964: "0.0.1"
```

Fig. 5: Sample document from the vulnerability collection

VI. Experiments

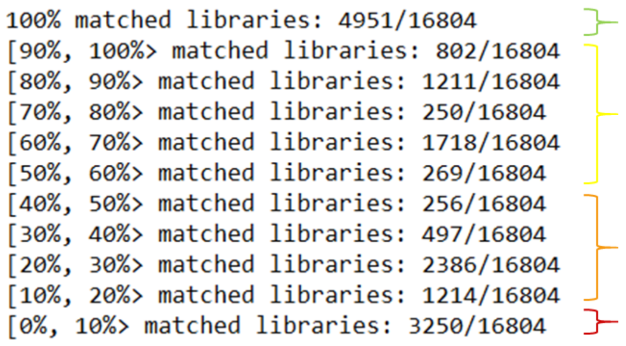
The first approach to version detection was to use hash values. The idea was to download as many source codes of libraries of different versions as possible and to calculate their hashes to store in the database. Retrieved hashes were used in comparison with hashes of unknown libraries. This approach did not succeed due to the small percentage of presence of original libraries. One of the reasons for weak hash detection is the ratio of the number of plugins to the number of original libraries. The number of plugins used on websites is significantly higher than the number of original libraries. It is very important to emphasize that at that stage plugins were not part of the database of known libraries. Another reason is source code that has been modified by a third party in order to personalize or change the functionality of the libraries themselves. The percentage of detected versions of unknown libraries using this method was approximately 1%, so it was necessary to find a new method of detection.

Greater success was achieved by combined detection using library-specific identifiers and differences between neighboring versions of known JavaScript libraries. The procedure consists of filling the MongoDB database with *new* and *old* lists and then comparing the unknown library with the list *new*. The goal is to determine the percentage of matches with previously known libraries.

After describing the detection procedure, the results and statistics of the obtained results is presented. The algorithm was run on 16804 unknown libraries and each one was detected with a percentage of confidence. Results are divided into four important groups through a more detailed manual analysis:

1) Version match (100% match)
2) Probable version hit (>=50% match)
3) Probable range hit (>=10%, <50% match)
4) Version miss (<10% match)

The results are viewed as satisfactory, as can be seen from the percentages of each group. From the results, it can be seen that in the "Version match" group, i.e. one hundred percent detection was achieved for a quarter of unknown libraries. The second group “Probable version hit” is also important. It mainly contains correctly detected versions of the libraries, but due to changes, they have a lower hit percentage. If the first two groups are considered as correctly detected versions, a hit percentage of about 50% is achieved. This means that for half of the unknown libraries, more precisely the JavaScript codes that were analyzed, the correct version is found. The next group is "Probable Range Hit". In this group there are libraries that were detected with a certainty greater than or equal to 10% and less than 50%. Due to the very structure of the vulnerability repository, it is not necessary to know the exact version of the library to detect vulnerabilities, but it is enough to determine a smaller range in which the exact version could be located. In our case, that range is the 10 most likely versions of the libraries that the algorithm returns as the best 10. About 20% of the libraries are in the mentioned group which, together with the previous two groups makes up more than 70% of all given libraries. Libraries from this group require further analysis if the correct version is to be discovered, as they are too modified to be discovered this way. In the last group, i.e. "Version miss", there are libraries that the algorithm could not detect, i.e. no information is known about their version. Manual analysis found that the JavaScript codes in this group are often not even JavaScript libraries, but ordinary JavaScript codes found on a web page. Detection statistics and their graphic representation are shown in Figure 6 and Figure 7.



```
100% matched libraries: 4951/16804
[90%, 100%> matched libraries: 802/16804
[80%, 90%> matched libraries: 1211/16804
[70%, 80%> matched libraries: 250/16804
[60%, 70%> matched libraries: 1718/16804
[50%, 60%> matched libraries: 269/16804
[40%, 50%> matched libraries: 256/16804
[30%, 40%> matched libraries: 497/16804
[20%, 30%> matched libraries: 2386/16804
[10%, 20%> matched libraries: 1214/16804
[0%, 10%> matched libraries: 3250/16804
```

Fig. 6: Detection statistics

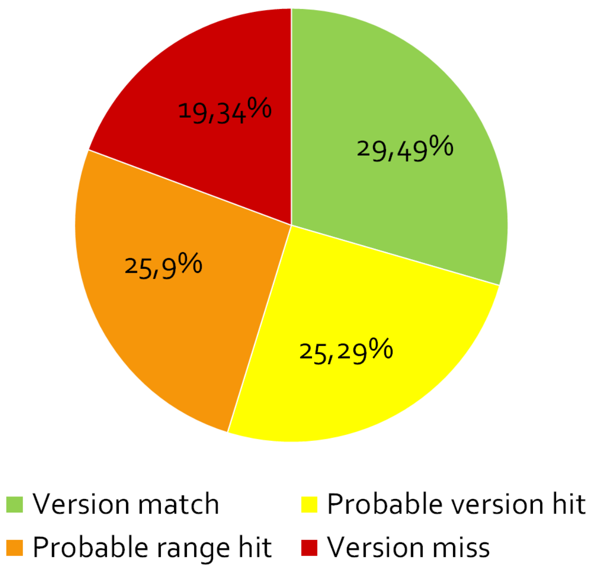


Fig. 7: Detection groups

VII. Discussion

Despite the relatively successful detection, the algorithm encountered some problems and limitations.

The algorithm itself is very complex. More precisely, its complexity is exponential $O(n^2)$. Even with additional optimization, the algorithm cannot be significantly speeded up on a large amount of data such as the database used in this paper. The greatest speedup is achieved by binding detection to identifiers, which reduces the range of libraries to compare. The range of libraries in the entire database, with slightly less than 300 different libraries and plugins, came down to exactly one library. Even after introducing this discovery step, execution time is still high. The algorithm took about 34 hours to discover 16804 unknown libraries, which gives us about 7 seconds per library.

One of the problems is the large number of unknown libraries that remain unknown due to the modification of the original libraries. They are made by developers in order to improve or adapt certain functionalities to their needs. To understand how much this affects the results, the fact that if the line differs by only one character, the line will not be detected is sufficient. Which results in a reduced hit percentage. Additionally, due to modification of the original library, it may happen that the vulnerability is fixed or a new one is introduced. If the vulnerability is fixed, detection will still report that the library is vulnerable. On the other hand, if the modification introduces a new vulnerability, due to developer carelessness or insufficient attention to security, detection will not occur. Some possible reasons for making changes are to delete parts of the code that will not be used and to add or rearrange code that complements and changes the functionality of the library to suit your needs. Also, one possible case is merging several libraries into one file in order to personalize it under its own name. An example of the modifications is the jQuery library [16], which contains more than 20,000 "forks" on its GitHub repository.

Another reason is that minified and obfuscated libraries are often used. As for minified, it's not that much of a problem because there are simple deminifiers. But if the library is obfuscated, such libraries will never be detected this way. There are very few of them, but they are present.

The next important problem that came after detecting a library version is determining vulnerabilities. There is no centralized database of known JavaScript library vulnerabilities. Even when the version is determined, if the library is not contained in the vulnerability repository it's not known whether it is vulnerable. This means that vulnerability detection is entirely dependent on the size and frequency of updating the vulnerability database. To identify vulnerabilities, the Retire.js repository was used but it contains only 26 of the most popular libraries. Therefore, a problem occurs when the detected library or plugin is not in the repository.

VIII. Conclusion & Future Work

When developing a software solution, it is very important to know potential vulnerabilities and to take care of the security of the program code at every step of development. Standard practice when developing any software solution is to use third-party libraries. The reason for this is to facilitate development if there is already a solution to some part of the problem and greater code security. That is why it is very important to monitor and update the versions of the libraries that the Web application uses in order to use the most secure version at all times.

The algorithm described in this paper automatically detects the version of an unknown library based on the library's source code. Furthermore, it can be used as a tool that analyzes large amounts of pages, looking for vulnerabilities that each page contains. Based on the detected vulnerabilities, warnings could be sent to Web site owners if they use a vulnerable version of the JavaScript library. Based on the detection results, it can be said that the algorithm detects the version of the unknown library very well. Nevertheless, this is only one technique of searching for vulnerabilities and should not be relied on alone but combined with already developed solutions.

Undoubtedly, there is a lot of room for improvement of the developed tool due to the limitations mentioned in the previous chapter. In order to improve detection, it is necessary to continue with new detection steps. Those steps should be even more detailed and continue detection of libraries that are not detected with 100% certainty. It is also possible to expand the database of known libraries with additional JavaScript libraries that are not currently in the database.

References

[1] Z. Bloom, "Javascript libraries are almost never updated once installed," Jan 2020. [Online]. Available: https://blog.cloudflare.com/javascript-libraries-are-almost-never-updated/

[2] Y. Wang, W.-d. Cai, and P.-c. Wei, "A deep learning approach for detecting malicious javascript code," *Security and Communication Networks*, vol. 9, no. 11, pp. 1520–1534, 2016.

[3] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 757–762.

[4] O. Tripp, P. Ferrara, and M. Pistoia, "Hybrid security analysis of web javascript code via dynamic partial evaluation," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 49–59.

[5] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Saving the world wide web from vulnerable javascript," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 177–187.

[6] M. Kluban, M. Mannan, and A. Youssef, "On measuring vulnerable javascript functions in the wild," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022, pp. 917–930.

[7] "Wappalyzer - identify technologies on websites," https://www.wappalyzer.com/, [Accessed 13-Jan-2023].

[8] "Retire.js - repository of known vulnerabilities." [Online]. Available: https://retirejs.github.io/retire.js/

[9] R. Bellairs, "What is static code analysis? static analysis overview." [Online]. Available: https://www.perforce.com/blog/sca/what-static-analysis

[10] J. Frankenfield, "Cryptographic hash functions: Definition and examples," Nov 2022. [Online]. Available: https://www.investopedia.com/news/cryptographic-hash-functions/

[11] "What is minification: Why minify js, html, css files: Cdn guide: Imperva," Dec 2019. [Online]. Available: https://www.imperva.com/learn/performance/minification/

[12] Beautify-Web, "Beautify-web/js-beautify: Beautifier for javascript." [Online]. Available: https://github.com/beautify-web/js-beautify

[13] [Online]. Available: https://www.cdnpkg.com/

[14] "Difflib - helpers for computing deltas." [Online]. Available: https://docs.python.org/3/library/difflib.html

[15] I. Kovacevic, M. Marovic, S. Gros, and M. Vukovic, "Predicting vulnerabilities in web applications based on website security model," in *2022 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. IEEE, 2022, pp. 1–6.

[16] "jquery," [Accessed 10-Feb-2023]. [Online]. Available: https://jquery.com/