Detecting Vulnerable JavaScript Libraries in Hybrid Android Applications

by

Nikita Volodin

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Smartphone devices are very popular. There are a lot of devices being sold, a lot of applications that are created and a lot of people using those applications. However, mobile applications could only be created in the native language of the mobile platform, for example, Java for Android, or Objective-C for iOS. The concept of hybrid mobile applications was introduced, in order to help developers create mobile cross-platform applications. These hybrid application platforms allow mobile application developers to use other development languages and their ecosystems. In particular, we are focusing on Android hybrid applications created using the Apache Cordova framework, which uses JavaScript (JS) as its development language.

We develop an automated method that can detect libraries used by hybrid applications that are known to be vulnerable. This search is performed by applying methods similar to Java code clone detection methods to the dynamic JavaScript language. We derive a signature from the reference JS file in a library and a signature from the unknown JS file in an application. Further, we compare those two signatures to produce a numerical similarity value indicating how close two files are. From this, we conclude whether the unknown file is identical or similar to the known reference file.

From the npm repository, we collect JS libraries that are known to be vulnerable based on the vulnerability data provided by Snyk, and end up with 10 686 distinct versions across 698 distinct libraries. We also have access to roughly 100 000 carefully collected Android applications, from which we find that 5652 are Cordova based hybrid applications. We find with manual verification for ten random apps that we can match 71% of library names and 80% of library names and versions. From the analysis of the entire application set, we find that 2557 (45.24%) hybrid applications from our reference set have at least one vulnerable library.

Our results show that it is possible to create a tool that conducts code clone detection for the dynamic JS language. Our approach still requires some refinement and improvements for minified JS files. However, it could be used as a stepping stone towards a very precise code clone detection tool based on JS source code analysis.

Acknowledgements

I would like to thank my supervisors Urs Hengartner and Mei Nagappan, for their guidance and considerable help with this research. Also, I would like to thank members of my thesis committee, Florian Kerschbaum and Michael W. Godfrey, for their time, valuable feedback and expertise.

This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo.

Dedication

To the ones I love.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Listings

Chapter 1

Introduction

Mobile and Web are two rapidly growing platforms. Both of these are used by a wide range of people for a wide range of tasks — from end users doing various things like gaming, movies, online banking, to businesses doing things like online communication and negotiation. It is frequently the case that one complements the other and when there is no mobile application to perform the task, there would be a Web application to do the same task. In order to help developers bridge the gap and build various cross-platform applications, the concept of hybrid mobile applications was introduced. These hybrid application platforms allow developers to use other development languages and their ecosystem in order to create applications that could be compiled into mobile applications for different platforms. In particular, we are focusing on platforms that allow applications built for the Web using JavaScript (JS) to be run on Mobile natively or very close to natively. The first generation of such applications for Mobile are called "hybrid" Web applications, and subsequent generations are called "progressive" Web applications and "truly native" Web applications. These applications are created using various Web frameworks, which allow compilation of existing Web applications into Mobile applications without the need for the developer to learn a new Mobile platform. For example, some of these platforms are Apache Cordova [9], other frameworks based on Cordova and React-Native [37]. We have selected to focus on the Web platform and JS, since JS is the most popular language on GitHub by the number of opened pull requests during 2017 [44].

Since developers can access the JS ecosystem, they can use a variety of libraries that are published on the de-facto standard registry npm [27]. These libraries enable developers to share and reuse code among different applications. Reusing libraries and sharing libraries on the central repository greatly helps developers. However, if a security vulnerability is discovered in one of the popular libraries, which might be in use by a lot of projects, all

these projects are potentially vulnerable as well. Because of this, if the dependent library is discovered to be vulnerable, there is an expectation for the developers of applications to update those libraries, when the patches are released. However, it is often not the case and there are a lot of projects using libraries that are outdated or known to be vulnerable.

While the popularity of these platforms is ever growing, the research dedicated to the security of these platforms is not comparable between the two platforms. For example, Android uses a very strongly typed development language — Java. The fact that this is a strongly typed development language allows for easier static analyses performed by various tools. For example, the software engineering community has produced a great variety of work related to the search of code clones in the Java code, detection of vulnerabilities, and others. Additionally, Android has seen security and privacy related research analysing various aspects and attack vectors of the platform. There are also papers proposing improvements and fixes in order to mitigate various issues. Additionally, there is work related to the mitigation and protection against intrusive Android components, such as advertisement related libraries. As for the Web platform, there is some vulnerability related research, but not as much research related to automatic detection of vulnerabilities, or research applying static analysis techniques to detect problems with the source code. For example, there is a wide range of papers analysing problematic components and finding vulnerabilities in browsers and server platforms, including suggestions to mitigate these problems. Most of these papers are looking at a specific problem in the platform and analyse whether a specific component is problematic. However, there are not a lot of tools applying static analysis techniques to automatically find issues related to various components of the Web platform. There is also not a lot of research from the software engineering community dedicated to dynamic languages such as JS, analysing issues such as detection of clones and others.

In this work, we develop an automated method that can detect libraries used by hybrid applications that are known to be vulnerable. This information could be used to either warn developers of those applications or the repository hosting those applications that some applications could be dangerous to their users. This search is performed by applying methods similar to Java code clone detection methods to the JS language. This task is often not easy in the case of JS code. That is because the source code of JS libraries could be modified or concatenated with other libraries. For example, it could be modified using the minification technique, which renames local variables and private functions throughout the code. Additionally, JS libraries could be bundled together in order to reduce the number of files transmitted over the network. This causes a lot of issues for previous detection techniques, such as the one described by Lauinger et al. [20]. These issues are further discussed in Section 2.3.

In this thesis, we are making the following contributions:

- We develop a method of creating a static signature of a JS file. The file in this case could either be a file for a single library, or a file for a bundle containing multiple libraries. Additionally the input file could either be minified or non-minified. Such a signature captures the essence of a source code functionality, while trying to omit some details that may add noise to the signature.
- We develop a method of comparing two signatures between each other. This allows us to match signatures of unknown files from applications to signatures of known files from reference libraries. Since the signature omits very granular details, the comparison between a signature of a minified and a signature of a non-minified file is also possible. The comparison also reports how close each particular signature to any other given signature is.
- We evaluate a large corpus of hybrid Android applications against all libraries published on the npm registry that are known to be vulnerable. We also evaluate the signature creation and the signature matching on a set of ten random applications from the large corpus.

We obtained 698 distinct libraries from npm, which correspond to 10 686 distinct versions across all libraries. In the course of the study, we discover that 5652 applications from around 100 000 applications use the Cordova framework. From applications that use Cordova, our tool finds that 2557 (45.24%) applications from our corpus have at least one vulnerable library, which was detected as the exact usage instance of a library. We report on the most used library names among the detected libraries and manually verify the top ten of the most used library names. We find that seven libraries were correctly detected, two libraries were not correctly detected, because they should have been filtered out at an earlier stage before analysis by the tool, and one is not correctly detected, because it seems like a false positive match by the tool.

We also perform a manual analysis on ten random applications, verifying results of the tool. Among the ten applications, we can manually find 17 different library usage instances, where the name of the library used in the application appears among the downloaded libraries, and 15 out of 17 have both name and version appear among the downloaded libraries. When considering only exact matches of name and version pairs of the library, our best matching metric detects 13 pairs in total out of which 12 are a subset of the 15 that it should have found. This gives us that the best matching metric has the precision of $12/13 = 92\%$ and the recall of $12/15 = 80\%$, when detecting library names and versions

and only considering the exact matches. We find that some of the libraries that were missed during the detection stage are missing because of optimization reasons and not because of problems with the algorithm.

Chapter 2

Background

This chapter discusses the background information about JS and hybrid Android applications.

2.1 JavaScript

JS is one of the most popular development languages. It was released in 1995 and has been used on the Web since. In 2009 the platform PhoneGap [43] was created by Nitobi and later in 2011 acquired by Adobe Systems. Adobe Systems released an open source version called Apache Cordova as one of the Apache Software Foundation projects. This platform allowed developers to write a single application in JS, which would be compiled into different mobile platforms, such as Android, iOS and Blackberry. Also, in 2009, the platform Node.js was released, which allowed developers to write server-side code using JS. Soon after, in 2010, the centralized registry of packages for Node.js, called npm, was created. In 2013, the framework Electron [14] was released, allowing developers to create desktop applications using JS. Later, in 2015, the framework React-Native [33] was announced by Facebook. Also, in 2015, the similar framework Native Script [3] was announced by Telerik. Both of these platforms allow developers to use JS to write mobile applications targeting multiple different platforms at once, while having improved performance compared to Cordova based projects. Originally the npm registry was only used to distribute packages for Node.js. However, with the development of new frameworks, developers started publishing and distributing libraries for those frameworks using npm. Thus, JS became useful not only to create web-sites, but also to create server-side applications, desktop applications and

native mobile applications. However, in this work, we only focus on using JS in the realm of cross-platform mobile applications.

2.2 JavaScript Libraries

Often developers distribute JS libraries with a common functionality for other developers to use in their projects. When used, these libraries have the same permissions as the code written by a developer. In other words, these libraries will have the same access to the web page and to the private resources of the web page as the source code of the web application. Because of this, if a library is compromised, the attacker via a compromised library can potentially steal the secret data of the web-page, such as cookies or other local secret tokens. Since the same JS libraries may be used during the mobile application runtime as on the web pages, similar attacks could happen against mobile users. However, the impact of such an attack might be higher in the case of mobile applications. That is because mobile application frameworks that are based on JS provide access to native Android or iOS functions via the JS interface available to any JS code running as part of the application.

Previous work by Lauinger et al. [20] has analysed the use of vulnerable libraries on web sites. In their work, the researchers collected popular libraries from many different sources, as sources might slightly alter the content of the source file. Further, they detected the use of vulnerable versions of those popular libraries on web sites. They employed two methods to perform this task. Firstly, they computed the hash of known JavaScript source files as well as the hash of unknown JavaScript source files and directly compared those hashes in order to detect exact copies of library files. Secondly, they relied on the libraries to identify themselves during the execution on the web page. For example, one library would export itself into the global scope under its given name and it would also export its own version as a property on the global variable.

However, there are difficulties with JavaScript, which can prevent those two methods to reliably detect unknown files as specific libraries. Namely, minification and bundling can prevent the hash based matching, and the need for a dynamic analysis as well as a lack of libraries reliably exporting their own version information can prevent the second matching approach.

```
var my_library_public_API = function () {
   var private_value = 42
   function checkValid(value) { return true }

   return {
      getPrivateValue: function () { return private_value },
      setPrivateValue: function (new_value) {
         if (checkValid(new_value)) { private_value = new_value }
      },
   }
}()
```

Listing 1: Example of the internal state and the public API for a library.

2.3 Difficulties with JavaScript Source Code

Since JavaScript is an interpreted language, the source code is a text file, which is executed by JavaScript engines. In the case of web development, the JavaScript source is transmitted over the network, and in order to reduce the transmitted size, the technique called minification is often employed. Additionally, in order to reduce the number of HTTP connections to the website, developers concatenate multiple different libraries together into one big bundle file. While these techniques are mostly beneficial for the web development case, they are still being used during mobile application development.

Minification is the process of reducing the size of the input source code without change to the functionality of the source code. There are multiple different tools that can perform minification on JavaScript source code. For example, tools like uglify-js [45], babel-minify [4], or google-closure-compiler-js [17]. In simple cases, these tools remove space characters needed for a human readable formatting of the code, as well as rename variables, functions and parameters, usage of which is known for the tool throughout the entire JavaScript program. More advanced minifiers can perform more advanced operations on the code without changing the behaviour of the program. For example, google-closure-compiler-js has a mode where it can eliminate dead code (unused functions or statements), and in some cases it can inline functions.

In JS, variables defined in the source code have a function level scope. That is variables defined inside functions are only visible inside those functions. This feature of the language allows for a simple encapsulation of the internal state, as seen in the example in Listing 1.

```
var my_library_public_API = function () {
   var private_value = 42
   function checkValid(value) { return true }

   return {
      getPrivateValue: function () { return private_value },
      setPrivateValue: function (new_value) {
         if (checkValid(new_value)) { private_value = new_value }
      },
   }
}()

var another_library = function () {
  function checkValid(value) { return value % 2 === 0 }

  return function (array) { return array.filter(checkValid) }
}()
```

Listing 2: Example of bundling two libraries using a function with the same name.

Using this feature, developers of libraries can easily expose the public API of the library without polluting the global scope with all of the internal variables or functions of the library. Since only a limited set of functions are exposed in the global scope of a given file, users of libraries can combine multiple libraries in the same file, without libraries conflicting with each other. The technique of combining multiple libraries into one file is called bundling. It is used to mitigate certain problems associated with a large number of different JS files included in the HTML page, such as a large number of opened HTTP connections by the browser. While there is no need for the reduced number of files in the case of hybrid mobile applications, this technique is frequently used by developers even for mobile applications.

As an example of bundling, given the code shown in Listing 1, we can add more libraries that are wrapped in the similar function exporting the public API of another library. Those additional libraries might use functions with same names but with different functionality, and these functions will not conflict with each other, since they will be encapsulated, as seen in Listing 2.

As an example of minification, we can see what happens to the code shown in Listing 2.

```
1  var my_library_public_API=function(){
2    var b=42;
3    return{
4      getPrivateValue:function(){return b},
5      setPrivateValue:function(a){b=a}
6    }
7  }(),another_library=function(){
8    function b(a){return 0===a%2}
9    return function(a){return a.filter(b)}
10 }();
```

Listing 3: Example of minifying the library using google-closure-compiler-js.

After using google-closure-compiler-js in `SIMPLE` mode, we get the minified code as output. For clarity, we add new line characters and indent some lines in the produced text, and show the example in Listing 3. As we can see, the minifier renamed all private functions and variables into one character names, since they are not exported and would never be accessed by the user of the library. Additionally, the minifier detected that the function `checkValid` defined on line 3 in Listing 2 is always returning true and the single place where it is being used on line 8 in Listing 2 is a conditional statement. Since this causes the if statement to always evaluate to true, the conditional is actually useless, so the minifier removed the check as well as the defined function.

This minifier has an additional `ADVANCED` mode. In this mode the minifier performs even more optimizations and minifications. For example, the code shown in Listing 4 is minified into code shown in Listing 5. Once again, we inserted new line characters in the minified code for clarity.

2.4 Hybrid Android Applications

Traditionally, Android applications are written only with the use of the Java programming language. Which means the business logic, the user interface representation logic, and libraries are written with Java. This feature, however, locks developers into using one language, which might have its own downsides.

One of the issues is that when developers attempt to capture as many users as possible, they need to cover different platforms. For example, for mobile applications, developers

```
const sum = function (a, b) { return a + b }

const fn = function (n) {
  let total = 0
  for (let i = 0; i < n; i++) {
    total = sum(total, i)
  }
  return total
}

console.log(fn(100))
```

Listing 4: Sample code for advanced minification.

```
for(var a=0,b=0;100>b;b++)a+=b;
console.log(a);
```

Listing 5: Example of minifying in `ADVANCED` mode.

need to create an Android application in Java, an iOS application in Objective-C, and native applications for other platforms. Developers could also choose to create a mobile web-site for major mobile browsers. In case developers also attempt to capture desktop users, they need to either create a web application in JS for all popular browsers, or create a native application for each major desktop operating system.

To mitigate the fragmentation issue, developers attempt to reduce the number of platforms they need to support. For example, developers of web sites can detect the mobile browser or the desktop browser and create a web site that adapts to the device. Using this technique, it is possible to create one responsive website, which will work on mobile browsers and on desktop browsers equally. While this feature reduces the number of different websites to create, it does not cover the realm of native mobile applications or native desktop applications.

Since web sites are ubiquitous, the Android and iOS mobile platforms provide a way to wrap a web page to be a part of a native application. For example, Android has `WebView`, iOS has `UIWebView` and other mobile platforms have similar technologies. Both of these features allow the developer to include HTML pages with JS code to control the representation of those pages, which essentially allows to distribute the web site as a native

application.

Because of this, the first generation of hybrid mobile frameworks was created. These frameworks allow the application developer to package an existing responsive website as a native application that runs in the WebView. These frameworks also take care of different mobile platforms, so that developers do not need to be aware of differences between them. Additionally, for security reasons, browsers do not provide access to certain OS functionality, such as the screen orientation, the sensor data, and others, unless the user grants an explicit permission. However, native applications might need access to information like this, so hybrid frameworks expose a public JS API, providing proxy access to certain Java functions.

Unfortunately, the WebView approach has downsides, which led to some developers not adopting this solution. Namely, the WebView approach is slow, compared to a native application written in Java. Additionally it is difficult for developers to manage transitions between different Android Activities.

These downsides led to the creation of a new approach, which is called "Truly-native" applications. These applications use a framework that does not use the WebView internally and attempts to solve issues with the previous type of hybrid applications. Just like the WebView based hybrid apps, these frameworks allow developers, who are already familiar with other languages, tools and ecosystems, to develop native applications without the need to learn a new ecosystem. Since JS is the most popular language according to Github [44], most of these frameworks choose to use the JS ecosystem.

These frameworks choose to avoid using the `WebView` or the `UIWebView`. However, they still use JS and still need to allow the developer to build applications that show some UI to the user. So, these frameworks take care of managing the UI of the application in the native language of the target platform and provide bindings for the JS code, allowing the developer to interact directly with native UI objects using the JS language. These frameworks also integrate the JS runtime engine with the native code, since the application logic is written with the use of JS.

2.4.1 WebView based Hybrid Applications (Cordova)

As mentioned above, the first attempt at hybrid applications was based on the `WebView` and corresponding technologies on other platforms. The very first framework that implemented this solution is 'Adobe PhoneGap' [35], which later was renamed and open sourced as 'Apache Cordova'. Since then, PhoneGap was based on Codova, as well as some other frameworks, such as Ionic framework [18].

The PhoneGap and Ionic frameworks build on top of Cordova. Both of these provide additional features as Cordova plugins, which Cordova does not have by default. However, both of these frameworks still use Cordova to create the mobile application. This means that for our purposes, applications created with PhoneGap or Ionic frameworks are indistinguishable from applications created with Cordova.

After decompiling a Cordova Android application with `apktool` [1], the web site, which is loaded into the WebView, is stored in the `/assets/www/` folder. It is possible for us to definitely determine that this particular application is using Cordova, because there will be a `cordova.js` file located in that directory. Also, since the source code for a Cordova based application has to be in a web site format, there has to be the main `.html` file, which defines all JS files used by the application and which is usually named `index.html`.

2.4.2 Truly-native Hybrid Applications (React-Native)

One of the early examples of such frameworks is React-Native, which is based on the front-end framework called React. At the time of writing, React is one of the most popular frameworks for the front-end development.

The React-Native framework integrates with the JavaScript source code better compared to Cordova. Hence, a React-Native application does not have an `index.html` file as an entry point. However, React-Native still ships the bundle of JavaScript code. The file containing all the business logic as well as libraries is located in the file on the path `/assets/index.android.bundle`. The presence of this file indicates to us that the application is created using React-Native and we can analyse it as a React-Native application.

Chapter 3

Related Work

There is a large number of papers that analyse various aspects of dealing with outdated or vulnerable dependencies, as well as about detecting code clones in source code and about various security aspects of JS security.

3.1 Dependency Management

In this section, we look at papers that deal with the management of outdated libraries and papers that deal with the management of vulnerable libraries.

Lauinger et al. [20] analyse the usage of JS libraries on the Web. They analysed over 133 000 websites and showed that 37% of them include at least one known to be vulnerable library. The paper also finds how the library was included — by developers, or by advertisement frameworks. The paper performs two aspects important for their task and interesting for us. One of them is identifying JS libraries and another is finding the source of vulnerability data. To identify the JS library, the authors used two approaches — static, where they hashed the source code and compared hashes, and dynamic, where they relied on the library to identify itself. The authors could not find a single definitive list of vulnerability data for libraries. So, they use sources that are either not JS specific or not organized in a machine readable format, and manually compiled the data from those sources.

Pashchenko et al. [34] perform a case study, investigating which vulnerable open source dependencies are actually used by software and which are not. That is they have the goal of reducing the set of vulnerable libraries by removing those that are unnecessary since no

one is using them. This helps with further steps of research or auditing, since there is no need to allocate resources to take care of unused dependencies, even if they are vulnerable. They come up with an approach and a tool, which measure and analyse Maven-based Java libraries to determine if these libraries or their dependencies are affected by a known vulnerability. The researchers perform an empirical study of 10 905 total versions across 200 Java open-source libraries and find that about 20% of dependencies that have a known vulnerability are not deployed. They also find that developers of libraries could easily fix 82% of their vulnerable dependencies. They define some dependencies as "halted" and characterize the usage of those libraries. A "halted" library is a library for which the next estimated release day, based on intervals between previous releases, is passed by a large margin. They find that under a conservative model characterizing those dependencies, around 14% of dependencies are "halted" and do not receive any updates.

Ponta et al. [36] introduce a new technique combining static analysis and dynamic analysis to determine the reachability of known vulnerabilities in open source libraries. Starting with the source code of an application, the researchers are recursively discovering all other methods used in libraries to find the set of functions that are reachable by the application. Since they know where exactly in the library the vulnerability is present, they can determine if the vulnerability is reachable or not. They also derive a method of suggesting to update a library, by informing the developer about the estimated effort and risk of updating the library.

Decan et al. [13] analyse the lag between the release of the latest version of some library and the time when the package using the library updates it. Such a lag is called the technical lag in the literature. While it is always desirable to have the technical lag equal to zero, it is not practically possible. The paper investigates the evolution of over 1 400 000 releases of 120 000 packages and 8 000 000 dependencies between these releases. The paper sets few research questions to investigate and to answer, among which, only few are of interest to us:

- "How long is the technical lag?"

  The paper finds that from 2015, the average technical lag for releases is between 7 and 9 months, where 25% of releases have a technical lag of more than 9 months, and 25% have a technical lag of less than 52 days.

- "How frequently are packages updated?"

  They find that on average, it takes between 12 and 22 days to update a release. Additionally it takes much longer to update a major version, compared to a minor

version, and it takes much longer to update a minor version, compared to a patch version.

- "How could technical lag be reduced by proper use of semantic versioning?"

  They find, that if dependency constraints were loosened and relied on the semantic versioning rules introduced by tilde ~ and caret ^ specifiers, "the proportion of releases suffering from technical lag could be reduced by 17.7%."

Mutchler et al. [25] evaluate the security of mobile applications that use an embedded web browser. The researchers develop highly scalable analysis methods to detect different classes of vulnerabilities, perform a large-scale analysis of Android applications to evaluate the amount of vulnerabilities, analyse the presence of these vulnerabilities and suggest changes to the Android API to mitigate these issues. They look at the extremely large set of 998 286 Android applications, which represents the complete snapshot of the Android Play market at that time, and focus on three vulnerabilities in mobile web apps — "loading untrusted web content, exposing stateful web navigation to untrusted apps, and leaking URL loads to untrusted apps." The researchers find that 28% of mobile web apps contain at least one security vulnerability. They also suggest a variety of measures to increase the security, from extensions to the Android API, to utilities helping the developer and utilities helping the user.

Our work focuses on hybrid Android applications, which are a mix of a mobile application for Android and a web application built using web technologies. While other papers have looked at those components separately, in our work we are focusing on implications of mixing these two technologies together. Similarly to Lauinger et al. [20], we have obtained a list of vulnerable libraries from which we are detecting libraries that are being used. However, in our case, we are employing a different and more robust matching method, in order to identify which library is being used.

3.2 Code Clones

The Software Engineering community has identified the problem outlined in this work and has already produced a significant amount of work related to the problem of code clones detection.

Sajnani et al. [40] present a new token-based clone detector, SourcererCC, which can detect clones at scale. The paper provides the definition of four different types of code

clones that are widely accepted by the community (the list is quoted directly from the paper):

- **Type-1(T1)**: Identical code fragments, except for differences in white-space, layout and comments.
- **Type-2(T2)**: Identical code fragments, except for differences in identifier names and literal values, in addition to Type-1 clone differences.
- **Type-3(T3)**: Syntactically similar code fragments that differ at the statement level. The fragments have statements added, modified and/or removed with respect to each other, in addition to Type-1 and Type-2 clone differences.
- **Type-4(T4)**: Syntactically dissimilar code fragments that implement the same functionality.

They come up with a code clone detector that can detect up to and including **Type-3** clones and that can work with projects of hundreds of millions of lines of source code. They come up with two different filtering heuristics that significantly reduce the number of comparisons between candidates, increasing the performance and allowing scaling. They also evaluate the performance of the new method compared to state of the art clone detection tools.

Davies et al. [12] introduce the general idea called 'Software Bertillonage'. The technique is employed in order to quickly and approximately reduce the large search space down to a manageable size, where it could be analysed more thoroughly with slower tools or with manual analysis. The paper introduces some definitions for Bertillonage Metrics and gives suggestions about how to measure those metrics. The paper also gives an example of how it could be applied based on a repository of Java libraries and performs a study evaluating the efficiency of the proposed metric for the case of Java.

Sabi et al. [39] attempt to find the impact on the detection of code clones caused by the rearranging of statements in a code block. They come up with a method of rearranging statements in the code block. With this method in place, they modify the source code of a project used in the experiment. They detect code clones using CCFinder in the unmodified source code and modified source code and report on the discrepancies between the number of detected code clones. They find that the number of code clones detected after the code was rearranged is very similar to the number of clones detected without the rearrangement of statements.

While there are a lot of papers focusing on Java, our approach is different in that it focuses on JavaScript. With the focus on JavaScript, we are addressing some of the specific

issues that are not present in Java code. For example, our approach is comparing libraries based on functions since the function is one of the smallest units of the functionality. In contrast, existing Java approaches compare libraries based on classes defined in a library since the class represents the smallest unit of the functionality in Java. Additionally, we are addressing features such as minification and bundling specific to JavaScript. These features are not common in Java and therefore existing tools might not be able to handle JavaScript code employing those techniques.

3.3 Vulnerabilities in JavaScript

There is a body of work that attempts to investigate whether the API available for web sites could be used with malicious intent. While these papers are not directly related to this work, these papers serve as a motivation for it.

Lekies et al. [21] investigate the usage of `LocalStorage` or `SessionStorage` for purposes of caching the source code of a JS application. They find that from the Alexa Top 500 000 Web sites, 20 422 Web sites use either `LocalStorage` or `SessionStorage` for data storage, and 386 Web sites store 2084 fragments of HTML, JS, or CSS source code in the client-side storage. They outline three types of attacks on the client-side storage, resulting in poisoned source code that is cached in the client-side storage. In short, the attacker can poison the cache in client-side storage in one of the three scenarios: 1. when the target website has a XSS vulnerability; 2. when the user is using an HTTP website on public/untrusted networks, *i.e.* in a coffee shop; 3. when a shared browser is in use, *e.g.* in an internet cafe or a public computer in a hotel lobby. They also propose a technique to mitigate the issue. In short, the server side calculates and stores the cryptographic hash of the code fragment to be cached. The client side calculates the cryptographic hash of the source code fragment when it retrieves the fragment from the cache and verifies it with the value returned by the server. The code fragment is verified and executed only when those two values match.

Son et al. [42] investigate the malicious usage of the `postMessage()` mechanism in `HTML5`. The `postMessage()` API allows for cross-origin communication of window objects, *e.g.* the main web-site window and a pop-up, or the main web-site window and an embedded iframe. Browsers set the correct value of the `origin` field, which the receiver of the message is expected to check and verify that it matches the expected value. They find that from the Alexa Top 10 000 Web sites, 2245 distinct Web sites use `postMessage()` functionality. From Web sites which use it, they only found 136 distinct `postMessage()`

receivers, because of high code reuse. From 136 receivers, they found that 65 do not perform any checks of the origin of the message, and 14 perform an incorrect check. Which leads to 84 Web sites having vulnerabilities, such as XSS and injection of content into local storage, caused by these issues. They also propose two defences, which could be deployed independently of each other. One of the defences proposes to use a pseudo-random token, which is given to the child frame during the creation time, implying that the attacker will have to guess the token in order to successfully communicate with the parent frame. The second defence is based on Content-Security-Policy which many browsers now support. They suggest extending the rules allowed by the policy to include a rule that controls messages between frames. If this extension is adopted by browsers, then using this HTTP header the browser will simply reject forbidden message sources.

Lekies et al. [22] present a system that can automatically detect and validate DOM-based XSS vulnerabilities. They designed and implemented taint-tracking in JS engines. To achieve this the JS engine string implementation was modified to include tracking bytes, which covers all features of the JS language and DOM API. They propose a novel and fully automatic way to validate vulnerabilities. They also create a sample exploit and verify that it was indeed executed. They analysed the Alexa Top 5000 Web sites and found that over 480 domains contain 6167 unique vulnerabilities.

Richards et al. [38] present a large scale study of the use of `eval()` in JS code, from the security perspective and the perspective of language designers. They collect usage traces of over 10 000 web sites and identify various usage types of `eval` functionality. They find that around 59% of web sites use `eval`. Although the number is high, at the time of writing, the usage of `eval` that was discovered by the paper was considered the best practice for certain use cases, such as backwards-compatible asynchronous data retrieval. Since then modern browsers started to provide features that achieve the same goals — asynchronous data retrieval — while being significantly safer. They also find that, while some usages were legitimate, more than half of the usages of `eval` were examples of misuse and up to 83% of the usages of `eval` could be rewritten using safer features of the language.

Li et al. [23] evaluate the usage of malicious redirections injected into JS code. Attackers are often able to find a vulnerable web-site and inject a malicious script redirecting the user towards the infrastructure of an attacker, where they can deliver malicious payloads to the victim. The paper investigates strategies that attackers employ in order to infect vulnerable web-sites to act as redirectors. They analyse 436 869 previously collected infected files and find that a lot of them are public JS libraries, and that the attacker tends to blindly inject malicious scripts in order to deploy them quickly. Using this information, they create a solution, called `JsRED`, which automatically detects unknown redirect scripts injected into publicly available JS libraries. They compare a modified version of the library with the

non-modified version to extract the difference and verify that said difference performs the redirect.

The approach that this paper uses in order to match an unknown/suspicious JS file to a known JS file is of a significant interest, since in our work we also have to solve a similar problem. First, they group all libraries by their filenames, as they are referenced on the website. This may result in large groups of candidates for each suspicious file. In this case, they need to quickly identify the reference file that is closest to the suspicious file. If those two are the same, then the suspicious file is the same as the library file and is not infected, so it does not need to be analysed any further. Otherwise, the approach extracts the difference for further analysis.

To quickly identify the similarity between two files, they employ two methods. One of them is a very quick approach to calculate the MD5 hash of both files and compare those hashes. This method will detect exact copies of files. The second method is slower but detects files when they were modified between each other. The paper proposes a solution based on matching the n-gram token sets derived from the unknown file and the known file. The solution splits files based on special delimiter tokens (*e.g.* `'`, `"`, `;`, `{}`, `()` and so on) into larger tokens, and over each larger token they pass with a sliding window of size 4 to extract n-gram token sequences. Further, they compare the *inclusion ratio* of tokens in the suspicious file to tokens in the reference file (quoted from the paper):

> "... given the set of n-grams for the reference R and that for the suspicious JS file S, we have their ratio $d(S, R) = \frac{|S \cap R|}{|R|}$."

In the end, for all the reference files, where the inclusion ration is higher than some threshold θ, they extract the difference for further analysis.

We have a very similar task to complete in our project. In our case, we also need to be able to search for two similar files, and have a numerical value showing how closely similar they are. We are using two metrics for our project, and one of the two metrics is the same as the one used in this paper. However, we are deriving a different method to create signature for two files that are being compared. In the paper above, the authors use an n-gram based search, while we create the token signature from the source file and compare the resulting token list.

All of the papers listed above are focusing on various aspects of the JavaScript language itself and how can those features be misused. In our work we are looking at a different threat area — issues that might come from dependencies included into the JavaScript-based project. For example, if the original application does not have a single vulnerability

and is not using a single feature that could be misused, it might still become vulnerable, if it will depend on a library with known vulnerabilities.

Chapter 4

Approach

4.1 Data Gathering

In this section we discuss the way we collect vulnerable JavaScript libraries as well as Android applications that we wish to analyse and attempt to detect vulnerable libraries in them. We also describe how we select and verify a small set of applications in order to evaluate the performance of the framework.

4.1.1 Collecting Vulnerable JavaScript Libraries

Traditionally, JavaScript libraries have not been distributed in a centralized way. Usually, the developers of a particular library would host their library on their official website, which would then be used during the development process. In the production mode, users of a library would either host the library on their own website, or they would utilize one of the CDN services hosting the library. However, since the emergence of JavaScript on the server-side, there was a need for a centralized registry of libraries, fulfilled by npm [27]. Over time, npm started being used as a distribution platform for libraries that are targeted for web sites, and later for libraries for mobile applications.

There are multiple projects on the Internet dedicated to tracking vulnerabilities reported for libraries distributed via npm. Two of these projects are Node Security Platform [26], which was acquired by npm, and Snyk [41]. While Snyk also tracks vulnerabilities of libraries for different languages and platforms, *e.g.* Java and Ruby, both of these projects

track JavaScript libraries distributed via the npm repository, which were the focus of our study.

We were able to get in touch with the people behind Snyk and acquire the snapshot of all vulnerable library names and version ranges tracked by Snyk as of May 2018. With the snapshot of the npm database as of the same month, we were able to obtain all versions existing on npm that match version ranges as reported by Snyk.

The npm registry started as an open source project, and it exported the database of all public packages published to npm. Despite npm being a commercial project now, the registry database of all publicly published modules is still available for anyone. We replicated the database, filtered all library+version combos that were released after May 2018, and we matched all library+version combos against the list of vulnerable ranges for each particular library, as reported by Snyk. After this, we ended up with 846 distinct libraries, which corresponded to 15 251 unique library+version combos.

Later, during preprocessing of all 846 libraries, we filtered some of those that did not fit our criteria. We found a list of the most dependent upon packages on the npm repository [32]. From the list, we manually classified the top one hundred libraries into the following categories: 'tool' (*e.g.* `babel-core`, `eslint`, and `gulp`), 'server-side' (*e.g.* `commander`, `express`, and `mongodb`), 'testing' (*e.g.* `chai` and `mocha`), 'uncertain' (*e.g.* `handlebars`, `less`, and `node-uuid`), and 'none-of-the-above' (*e.g.* `core-js`, `jquery`, and `underscore`). Then, from the list of 846 libraries we excluded those that were classified into the first three categories, and kept those belonging to the other two categories. Next, during the creation of library signatures as described later in Section 4.2.2, we found that we cannot reliably extract the file that corresponds to the content of the library. We found that some developers are releasing the package with the main file loading the actual source code of the library, rather than directly having the main file as the source code of the library. In the case of these libraries, the created signature would be of zero length, because such loader files would usually not have any functions in them. We also found that for some libraries we failed to create the signature. Those libraries either had an improper format when they were packaged, or their main file could not have been parsed using the JavaScript parser we used. In summary, we excluded 3747 distinct versions with an empty signature, additionally, 693 distinct versions were excluded by blacklisting libraries from the top 100 list, and lastly, 125 failed to get analysed, so they were also excluded. After excluding all of the above, we ended up with 698 unique library names, which corresponded to 10 686 library+version combos.

4.1.2 Collecting Android Applications

From the prior work done by Angshuman Ghosh, Cassiano Monteiro, and Lakshmanan Arumugam [2, 10, 11] an extensive data set of Android applications was collected. It is a set of around 100 000 Android applications scraped from the Google Play Store, which we wish to analyse. The first step is to decompile each application to see the resources that it has and to detect if it is created using Cordova or not. We do so with the use of `apktool` [1]. Since applications are created and packaged using different frameworks, applications will have a specific structure, which is unique to the given framework. For example, Cordova based hybrid applications will have an `index.html` main file and a `cordova.js` file beside the main HTML file. In the case when these files are not present in the decompiled application, we deem this application as not Cordova based, and we remove it from the further analysis.

After filtering all applications and keeping only Cordova based ones, we ended up with 5652 Android applications.

4.1.3 Manual Analysis of Ten Apps

In order to verify our approach and the tool, we chose a small subset of applications that we manually investigated. We selected nine random applications from the filtered set of 5652. We also found one Cordova based application from the showcase section of the official Cordova website, which we added to the set. We manually analysed these ten Cordova based applications and compared the results of this analysis to the results produced by our tool.

At first, we prepared these ten applications by decompiling each one, inspecting the entry file `index.html` and finding all JavaScript files that were used by the application. We categorized each file into one of four categories: 'business-logic', 'http-script', 'bundle', and 'single-library'. The scripts referenced via http/https link do not have source code present in the application and therefore we are unable to analyse them either manually or with the framework. Additionally, previous work by Lauinger et al. [20] focuses on the use of libraries over the web, and hence their work could be applied in this case. Scripts determined to be the business logic are performing the primary functionality of the application, and hence they should not contain any libraries.

The files of a particular interest for us are the ones that we categorized as 'single-library' or 'bundle'. These files contain libraries, which we can detect manually and can compare with framework results. Files categorized as 'single-library' contain only one library file

in them and 'bundle' files contain multiple libraries concatenated together into one bigger file.

As mentioned above in Section 2.3, JavaScript source code is essentially a text file, and it can be minified with specific tools without the loss of functionality. Because of it, some meta-data in the file about the library might be lost. For example, comments might be stripped away and some private variable names that are unique to the library could be renamed into generic variable names that do not convey any information about the original library. Because of this issue, we are also recording whether the detected library in the application file is not a guess because of comments throughout the library or a guess based on the structure of the file. In detail, if the file has a comment that defines the name of the library that is being used, then we record the particular usage instance as a non-guess. Otherwise, if we are able to detect the library in use by looking at the file name, at the names of variables, or at the constant values used throughout the source code, then we record the usage instance as a guess. However, if we cannot identify the library as either a guess or a non-guess, we omit the library from the manual analysis. Similarly, it is not always possible to detect the exact version of the library that is being used, even if we are able to detect the name of the library.

Throughout ten Cordova applications, we found 63 files that correspond to either 'single-library' or 'bundle'. Among these files we manually identified 106 distinct instances of various libraries in use. These usage instances were either a guess or a non-guess, and for some of them we were only able to extract the name and unable to extract the version information. Out of 106 total instances of library usages, we detected 56 as matches that were not guesses. Then from those 56 instances we excluded matches that were not found in the Snyk vulnerability database. We ended up with 17 usage instances where the name appeared among the downloaded vulnerable libraries, out of which 15 had both the name and the version appear among the downloaded vulnerable libraries. The two libraries that we manually detected, but which we do not have among the downloaded vulnerable libraries, were not downloaded because they are no longer available on the npm registry.

The names and versions of all applications are given in Table 4.1 with all 17 usage instances of vulnerable libraries. The two instances that do not have the version appear among the downloaded vulnerable libraries are marked with the star ('*'). Additionally, not all applications, out of the ten we selected in the beginning, had instances of vulnerable libraries being used. In our case, it was only eight applications across which 17 usage instances of vulnerable libraries were detected.

Table 4.1: Usages of vulnerable libraries.

Application ID (Version)		Library@Version
br.com.williarts.radiovox (20008)		`angular@1.5.3` `moment@2.17.1`
com.atv.freeanemia (1)		`jquery@1.11.1`

com.paynopain.easyGOband (18)		`angular@1.4.3`
		`jquery@2.1.4`
		`moment@2.9.0`

com.tiny.m91392d54e89b48a6b2ecf1306f88ebbb (300000016)		`angular@1.4.3`
		`lodash@2.4.2`
		`jquery@2.2.2`

com.tomatopie.stickermix8 (10)		`jquery@1.11.2` `bootstrap@3.3.2`
io.shirkan.RavKav (1800000)	*	`jquery@2.1.1` `jquery-ui@1.11.1`
net.jp.apps.noboruhirohara.yakei (102008)	*	`jquery@1.9.0`

it.wemakeawesomesh.skitracker (102081)		`jquery@2.1.1`
		`angular@1.3.0`
		`moment@2.8.3`

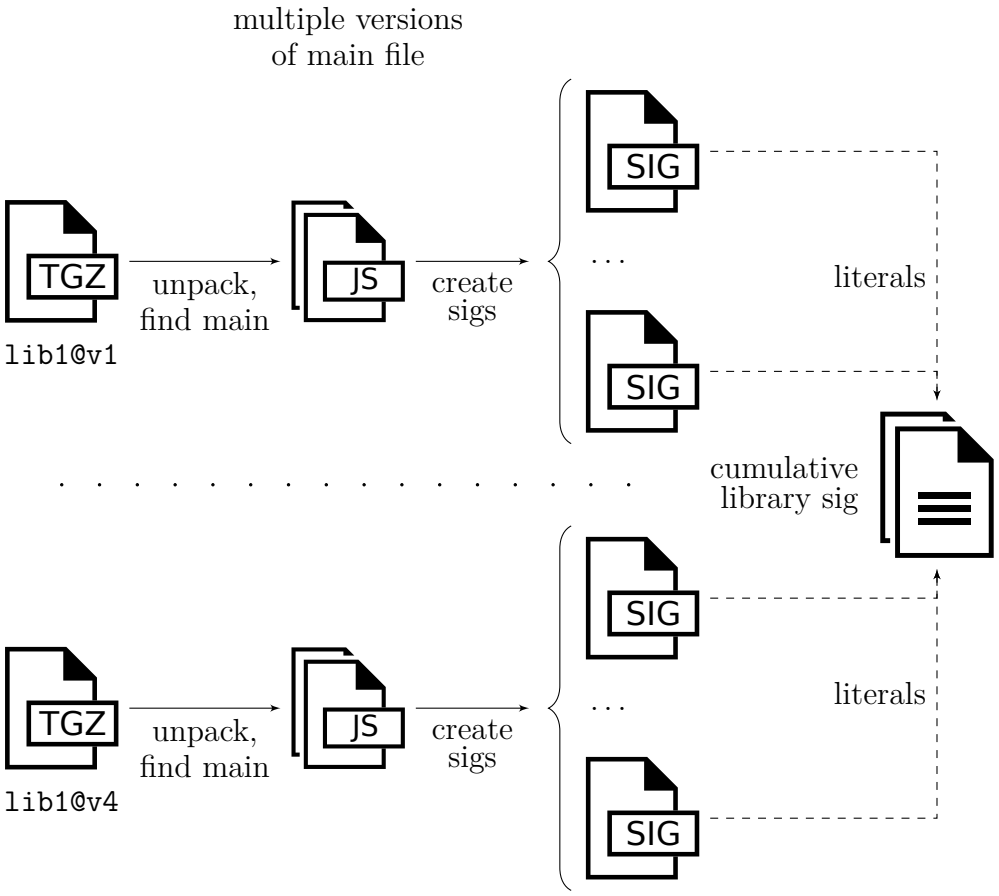

Figure 4.1: Workflow of the first stage, preprocessing the library.

4.2 Algorithm

We split the analysis into separate stages, so that later stages could be run separately as many times as needed, without unnecessary re-computation of previous stages. In the first stage, which is illustrated in Figure 4.1, we create the signature for all JavaScript files in all libraries that we wish to detect among applications. The set of libraries could be selected based on any criteria, but in our case, these are libraries available on the npm registry that are known to be vulnerable. During this stage, we both create a signature for each specific version of a library, as well as a cumulative signature of all versions of one library. In the second stage shown in Figure 4.2, we create signatures for all JavaScript files among all Android applications. We also create an initial list of candidate library names

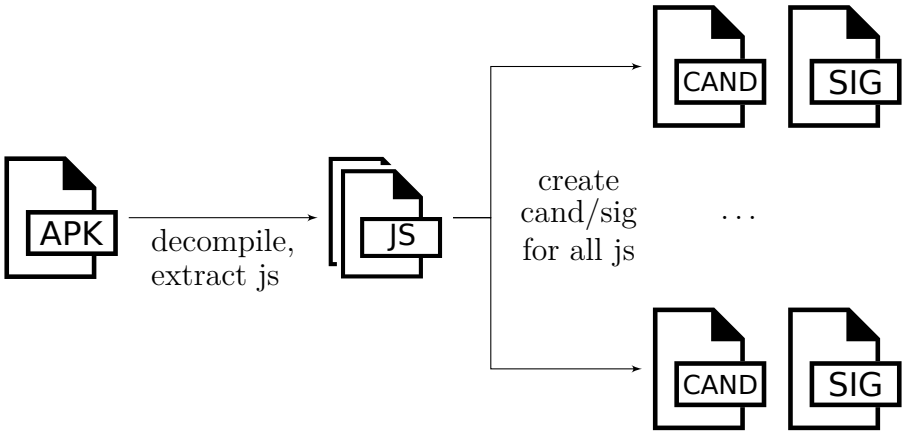

Figure 4.2: Workflow of the second stage, preprocessing the application.

for each file, based on the cumulative signature of all versions of a given library. The list of candidates is needed to significantly improve the performance of the entire search. For example, our dataset has 698 distinct libraries downloaded, and with the list of candidates in place, we will only need to look at the top ten most likely libraries. It is also important to note that every single JavaScript file, irrespective if it originated from a library or from an application, undergoes the same procedure of the signature creation, meaning any two signatures will have a similar semantic and they could be compared. In the third stage shown in Figure 4.3, we compare an unknown signature of files from Android applications against known signatures of files of all versions of libraries, that we identified as candidates during the previous stage.

4.2.1 Create the Signature of a JavaScript File

For every JavaScript file found by the tool, the signature will be created following the method described below. The signature is created in a deterministic way, so if two files are the same, they will have exactly the same signature created. Additionally, the signature is created based on the content of the file and it captures the actual source code as it is created by the developer.

The signature for one file consists of two distinct parts, representing various information about the file. The first part of a signature is a list of all literal values that are encountered in the file. The second part of a signature is a list of function signatures for all functions encountered in the file. To create both of these signatures, we parse the JavaScript file

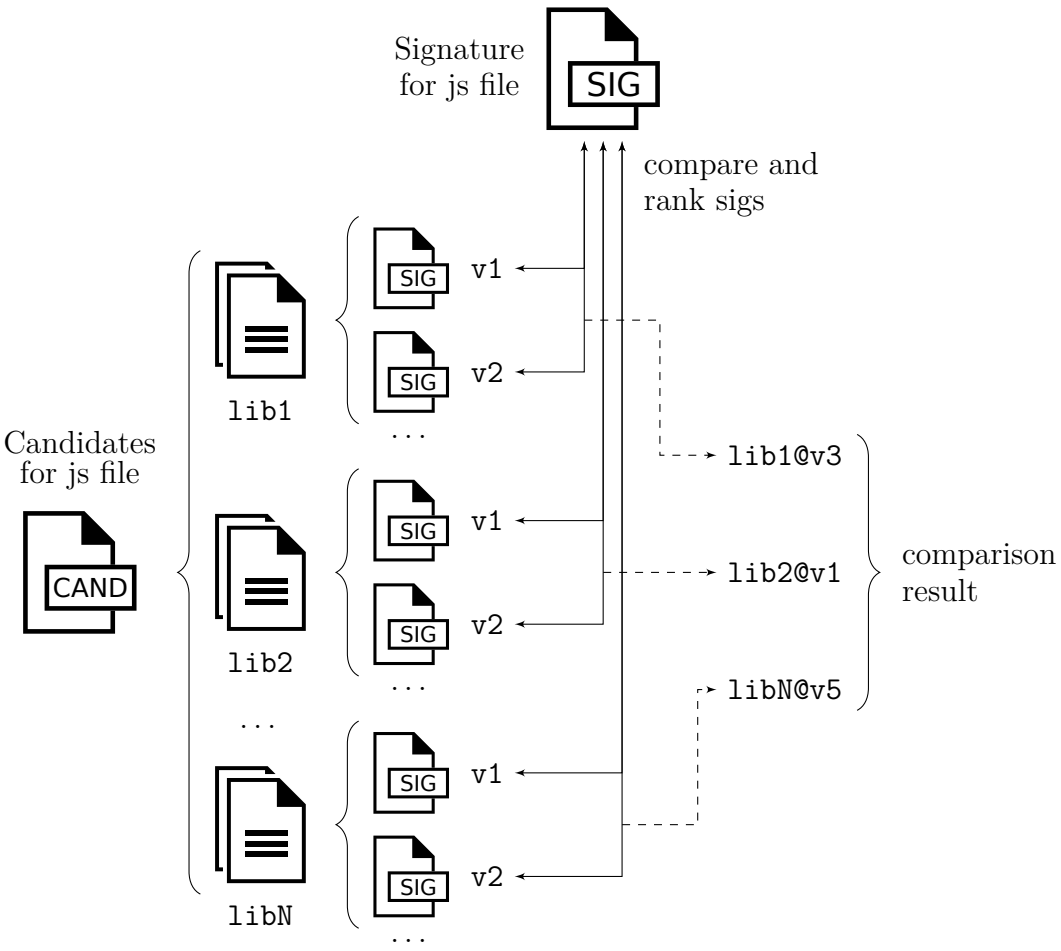

Figure 4.3: Workflow of the third stage, comparing signatures.

with a parser and create an Abstract Syntax Tree (AST) for the source code, which we traverse to discover nodes corresponding to functions or nodes corresponding to literal values. Parsing the JS source code and creating the AST is not an easy task, so we use one of the existing and well known JS parsers — `babylon` [5] — to get the AST.

In JS, there are only few literal values: String/Template, Number, Regexp, Null and Boolean. These literals correspond to constant values hard-coded by the developer in the source code. These values should be present in any library, since most libraries will have at least error messages, which are represented as String literals. It is also obvious that some of these values will be used abundantly and will not actually bring any valuable information about the file. For example, when for-loops are used, it is very common either to initialize the counter at `0`, `-1`, or `1` and count up, or to initialize it at a high value and count down until 0 or some other low value. Also, as a second example, it is quite common to use the empty string as a string initializer, so this value will also be very common and will not bring any important information about the file. Because of these reasons, we exclude integer numbers in the range from `-1` to `5` inclusively, empty strings, Null literals and Boolean literals.

The second part of the file signature is created based on functions that we encounter during the traversal of the AST. Since functions could be defined inside other functions in JS, for every function discovered in the source file we record the parent–child relationship between all of them. Thus we end up with the tree of function nodes, which we then flatten out, while keeping the trail of parent–child relationships between functions by the matter of names of functions. For each function that we discover in the AST, we extract the following information about the function, which we consider the most important information.

- The name of the function
- A list of types for each statement in the function
- A list of tokens for each statement in the function

In more detail, the name of the function is extracted during the traversal of the AST. In JS, functions can have from zero to two names. There are functions that could be declared with the name (as seen in Listing 6 on line 1), or they could be declared without the name, in which case they become anonymous (lambda) functions (as seen in Listing 6 on line 2). While anonymous functions could not be accessed by developers, they are commonly used when the function is requested as a parameter for a different function, or when these anonymous functions are assigned to a variable. There are also functions that could be declared without a name, but they are assigned to a variable (as seen in Listing 6 on line

```
1 function a() { return 1 + 2 }
2 function () { return 1 + 2 }
3 var b = function () { return 1 + 2 }
4 var c = function d() { return 1 + 2 }
```

Listing 6: Function naming examples.

3), and there could be functions that are declared with a name and which are assigned to a variable (as seen in Listing 6 on line 4). In the case of two names for the function, we choose the name defined during the declaration of the function, not the name of the variable to which the function is assigned. Additionally, while in cases of a named function declaration the AST reports the identifier name without a problem, for cases when the function is assigned to a variable, there is no easy way to access the name of the variable, when we discover the function node. For this case, we have to perform a look-ahead in order to detect function nodes further in the tree and record the variable name as a possible function name. Lastly, in those cases where we cannot discover the name of the function, we set the name as the constant value `'[anonymous]'`.

The list of types for each statement in the function is the second data piece that we extract from the function. This list is one of the two lists that are based on the content of the function. The AST parser that we are using produces the list of nodes for each statement in the function. Each of those node objects has a type associated with it. Because of this, it is very easy for us to extract types of statements in a function — we simply iterate over the list of objects and extract the type for every element.

The list of tokens for each statement in the functions is the third data piece that we extract from the function. This list is the second one that is based on the content of the function. As mentioned above, the parser returns the list of nodes, where each node represents one statement. Each statement is constructed of other statements or expressions, and each expression could be constructed from more expressions. So, for each statement type we extract relevant information to capture the essence of a given statement and then recursively parse child statements and expressions that were used as a part of the parent statement. When we encounter the child statement, we pass it to the same function that parsed the parent statement. This way we again will only extract relevant information for the given child statement and its children. When we encounter the child expression, we pass it to a very similar function that parses each expression and extracts only relevant information from the given type of an expression. However, if we discover a statement or an expression related to the function declaration, we will only extract the information

about the function name that is being called or created. We will not recursively parse the function itself, since in our previous step we already extracted all functions and we know that the encountered function will undergo the process described here at some point. From all recursive traversals we extract relevant information, which we call tokens. These tokens are collected together into one single list and stored as a list of tokens in the signature of the parsed function.

We show an example source code and its extracted signature in Listing 7. In the listing, there are three functions defined on lines 2, 3 and 7. For these three functions, there are three objects produced in the array with the signature — on lines 1, 8 and 15.

The function defined on line 2 is anonymous and is assigned to the variable with the name `b`, therefore the function receives the name of that variable. In the function body, there are two statements defined — the first one corresponds to the function declaration on lines 3–6, and another one corresponds to the function call on lines 7–9. The first statement is declaring the named function with the name `fn1`, and it has the type `FunctionDeclaration`. It is simple to save types of statements in the signature, therefore in the resulting signature, the extracted type is `FunctionDeclaration` as shown on line 3. However, for the tokens we have to select what to save and not. In the case of `FunctionDeclaration`, we are saving the name of the declared function, if we can get it. Therefore, on line 6, we can see the resulting token corresponding to this `FunctionDeclaration` statement. The second statement corresponds to calling the function with parameter `a`, and it has the type `ExpressionStatement`. Because of this, the extracted statement type is `Expression` as seen on line 4. As for the tokens, for the `ExpressionStatement` type, we chose to save the function that is being called and values that are passed to the function. Because the function does not have a name and we are passing the variable `a` to it, the resulting tokens, shown on line 7, include information that the `anonymous` function is being called with the variable `a`.

Both of the functions in the source code on lines 3 and 7 are defined inside the first function on line 2. Because of this, extracted names for these two functions are prefixed with `b:»:`, which correspond to the name of the surrounding function. The function declared on line 3 has the name `fn1` assigned to it. It contains two statements — the first one declaring the variable and assigning the value to the variable, and the second one returning the variable. The first statement has the type `VariableDeclaration`, which is extracted into the signature and shown on line 10. When we are creating tokens for this statement type, we chose to save the created variable name and the type of the assigned value. However, if the variable does not have an initializer, then we will skip saving the variable into the signature. The resulting token for this type of the statement can be seen on line 13. The second statement has the type `ReturnStatement`, so we are saving `Return`

```
var a = '1';
var b = function () {
   function fn1() {
      var c = '1';
      return c;
   }
   (function (param) {
      return param + '2';
   })(a);
}
```

```
[{ name: "b",
   types: [
     "FunctionDeclaration",
     "Expression" ],
   toks: [
     "Function[fn1]",
     "Call[Function[anonymous](a)]" ]},
 { name: "b:>>:fn1",
   types: [
     "VariableDeclaration",
     "Return" ],
   toks: [
     "Variable[c = String]",
     "Return[c]" ]},
 { name: "b:>>:[anonymous]",
   types: [
     "Parameter",
     "Return" ],
   toks: [
     "Return[Binary[param + String]]" ]}]
```

Listing 7: Signature extraction example.

into the signature as a type, as seen on line 11. As for the tokens for this statement type, we chose to also save the name of the returned variable, as shown on line 14.

The last function on line 7 is defined without the name, so we assign it a dummy value `[anonymous]`. It also has a parameter declared in it, with the name `param` and it has a return statement, that returns the binary expression concatenating two elements. Because of this, statement types are extracted as `Parameter` and `Return`. However, when we are extracting tokens, we chose to skip parameter declarations, since they could be easily minified. So, we only extract tokens that correspond to the statement on line 8. Since the `ReturnStatement` returns the binary expression concatenating two values, we record the binary expression as well, as shown on line 20.

4.2.2 Stage 1: Create Library Signature

Before this stage, we already acquired all versions of all libraries that we wish to detect among our applications. These library+version combos are downloaded from npm as a tarball file and stored on the disk. All libraries coming from npm have a specific and distinct structure. Namely, the content of the package is stored in the folder `package/` inside the tarball and always has the npm specific metadata file `package/package.json`. However, we discovered that there are different tools that could be used in order to package the library to distribute using the npm repository. Each of these tools has its own metadata file containing the location of the main entry file of the JS library. We identified three main tools for packing the source code for distribution over the npm registry.

1. The `bower` [7] package manager.

   The main metadata file is `package/bower.json`. This package manager allows to specify multiple JS files. The files are specified in the `main` field of the metadata file.

2. The `componentjs` [8] package manager.

   The main metadata file is `package/component.json`. This package manager allows to specify multiple JS files, like `bower`. The files are specified in the `scripts` field of the metadata file.

3. The `npm` package manager.

   The main metadata file is `package/package.json`. Unlike `bower` and `componentjs`, this package manager allows to specify only one main JS file. The main file is specified in the `main` field of the metadata file.

If the library was packed with either `bower` or `componentjs`, the resulting tarball will have two metadata files. One of the two metadata files corresponds to the file for the specific tool, containing most of the information, and another file is for the npm registry, allowing the package to be distributed over npm. In this case, only the metadata file corresponding to the tool will contain the field pointing to the main JS file of the library. Because of this, when we are searching for the main file of the downloaded library, we are first trying the `package/bower.json` file and then the `package/component.json` file. If none of those two files exist, then we are trying to extract the location of the main JS file from the `package/package.json` file belonging to npm. Lastly, if none of the previous methods work in the specified order, we try a few hard-coded paths in case the developer of the library followed some common pattern. We try to detect if there is a file located at any of the following paths:

- `package/dist/name.js`,
- `package/dist/name-version.js`,
- `package/name.js`, or
- `package/name-version.js`.

If none of the methods listed above successfully extract the main JS file, we deem the library as failed to be analysed and we skip it. However, if we successfully extract the main JS file, we also attempt to extract a minified version of the same file by trying the common pattern for storing minified files. We are adding `.min` into the filename and checking if there is a minified file present at the given path. For example, if we found that the main file for the library is located at the path `package/dist/jquery.js`, then we verify if the file `package/dist/jquery.min.js` exists, and use that as a minified file for the discovered main file.

This extraction method assumes that the main file, as indicated by the metadata file, is the one that contains the entire functionality of the library. However, there are cases when this is not true. For example, sometime the main file indicated by the metadata file is merely a loader for the actual source code of the library. As of now, our tool is not able to handle this case. However, it should be possible to implement by following all import references declared in the main file and in all respective files loaded by the main file. These loader files are distinguishable from actual library files, because loader files usually have one or no functions defined in them, and therefore the signature created for such files will be empty or only containing one function.

After we successfully extract one or more main files — there could be multiple files specified by the main field of various metadata files, as well as there might be additional minified files — we create the signature for all of these files, as specified above in Section 4.2.1. Additionally, we create the aggregate (or cumulative) signature for the entire library, using signatures created for all different versions and variations of JS files of the given library. As mentioned above in Section 4.2.1, we extract all literals that we encounter in the JS file. We collect all literals for a specific version and create the unique set of these literal values for one version of the library. Next, we create a union with unique sets of other versions of the same library. The final combined union of unique literals represents the signature for all versions of the same library. This additional signature of the entire library helps us to increase the performance during the subsequent matching.

4.2.3 Stage 2: Create Application Signature

Before this stage we already decompiled our applications and removed all those that are not Cordova based applications. Since only Cordova applications are left, we know that there is a main `index.html` file present and the location of the file. We load this file and parse it using the HTML parser. From the file, we extract all script tags that appear in the `'<head>'` and `'<body>'` section of the HTML document. Script tags will either contain the source code as the content of the script tag, or it will point to the JS file using either an HTTP URL or a file URL. We can proceed in the case when we have the source code available — either a script tag with the content, or a link to a file existing in the application APK. For every extracted script file, we create the signature as described above in Section 4.2.1.

Additionally, we create a list of candidate library names for every file in the application. At this point we already have created a signature of the application file and aggregate signatures for all libraries. In order to determine if the library is a candidate for the given file, we are comparing the literal part of the file signature against the aggregate literal signature of a library. There are two methods that we employ in order to define a library as a candidate, where the first one takes priority over the second one, and the second one is only used if the first one cannot identify any candidate libraries.

The first method detects if the literal signature of a given application file is a subset of a cumulative library signature. If it is the case then this library is considered a candidate for the file of the application for further search. Across all downloaded libraries, this method will detect usages of a single library in the application file. However, it will fail in the case of a bundle file in the application. That is because literals in a bundle file will never be a

subset of any single library. In this case, the candidate creation falls back to the second method. In the second method, we are comparing the literal signature of an application file against the cumulative signature of a library using the Jaccard metric described later in Section 4.2.4. This metric defines how one set is similar to another set. After computing this metric for all libraries between the application literal set and the cumulative library literal set, we are ranking candidate libraries based on the value of the Jaccard metric. After we ranked all of the libraries, we take the top ten libraries and use those libraries as candidates for further analysis.

4.2.4 Matching the Unknown File Signature to the Known File Signature

In order for us to perform the next step of the matching algorithm, described later in Section 4.2.5, we have to be able to compare any two file signatures. The algorithm relies on the existence of functions that implement the interface defined by the `COMP_TYPE_FN` function in Algorithm 1. The `COMP_TYPE_FN` interface compares signatures of two files — the signature of an unknown file (u_sig) to the signature of a known library file (l_sig). The implemented function has to return the similarity value between signatures in the range $[0, 1]$, as well as a mapping between matched functions of the two signatures passed to this function.

Algorithm 1 Compare two file signatures

1: **function** COMP_TYPE_FN(u_sig, l_sig)
2: $similarity \leftarrow [0; 1]$
3: $mapping \leftarrow Map\{u\_sig\_i \Rightarrow l\_sig\_i\}$
4: **return** $(similarity, mapping)$
5: **end function**

As one of the basic and low-level tasks of comparing two file signatures, we need to be able to compare the similarity of two lists[1] between each other. For example, given two lists of function names, we would like to know how close these two lists are to each other, based on the number of identical function names defined among both lists. One of the metrics that fits into this requirement is Jaccard Index [19]. This metric produces the value as a quotient of the size of the intersection of two sets over the size of the union of

[1]Here and further in the discussion of indexes, the term 'list' refers to multiset, as the ordering of elements does not influence the value of an index.

those two sets. For example, given two identical sets, $A = \{a, b, c, d\}$ and $B = \{a, b, c, d\}$, the Jaccard Index is equal to 1.

$$J = \frac{|A \cap B|}{|A \cup B|} = \frac{|\{a, b, c, d\}|}{|\{a, b, c, d\}|} = 1$$

Unfortunately, this method is defined only for sets of values. Since the names of functions in JS could repeat, this method does not work very well for the lists of function names. So, we create a modified version of this metric, which takes repeated elements into account.

The modified version of the Jaccard Index, called Jaccard Like Index in the following text, goes through elements of the first list, and if it can find a matching element in the second list, then this element is added into the intersection list, and removed from the second list, and if it cannot find a matching element in the second list, then this element is stored for the further calculation of the final value. Then, the final similarity value is calculated as all elements that were seen in both lists, over the concatenation of three lists — the intersection list, the list of leftover items from the first list and the list of leftover items from the second list. This method ensures that we can find multiple repetitions of the same element in both lists. For example, given two lists, $A = [1, 2, 2, 3, 3]$ and $B = [2, 2, 3, 4]$, the Jaccard Like Index is equal to 0.5, and the value of the original Jaccard Index is equal to 0.5.

$$J_{like} = \frac{|A \mathbin{\widetilde{\cap}} B|}{|A \mathbin{\widetilde{\cup}} B|} = \frac{|[2, 2, 3]|}{|[1, 2, 2, 3, 3, 4]|} = \frac{3}{6} = 0.5$$

$$J = \frac{|\{2, 3\}|}{|\{1, 2, 3, 4\}|} = \frac{2}{4} = 0.5$$

Even though in this example the original Jaccard Index for sets has the same value as the Jaccard Like Index, the modified version gives the benefit for skewed lists. For example, given two lists, $A = [1, 2, 3]$ and $B = [1, 2, 3, 3, 3, 3]$, the original Jaccard Index will give the value of 1, but the modified Jaccard Index will give the value of 0.5.

$$J_{like} = \frac{3}{6} = 0.5$$

$$J = \frac{3}{3} = 1$$

It is also worth noting that these two metrics do not make a distinction which one of the two sets is known or unknown. In other words, regardless whether we assume that the list

A corresponds to the unknown signature and the list B corresponds to the known signature or vice versa, we will get the same similarity value.

Additionally, we define another pair of metrics, which are slightly simpler than the Jaccard Index or the Jaccard Like Index. As mentioned above, both the Jaccard Index and the Jaccard Like Index do not make a distinction between the order of the two sets. However, since we know that we will always compare an unknown signature to a known signature, the two new methods assume that the first/left signature corresponds to an unknown signature and the second/right signature corresponds to a known signature. Since we know which signature is which, we define the metric as the proportion of the same elements between two signatures to the total number of elements in the known signature list. Analogous to the original Jaccard Index and the modified Jaccard Like index, we have two versions of this metric. One of the two versions is assuming sets of elements passed to it and is called Our Index (for legacy reasons). Another version is allowing lists to be passed to it and is called Lib Portion Index . As an example, given two lists $A = [1, 1, 2, 2, 3, 4]$ and $B = [2, 2, 3, 4, 4]$, the values of the metrics are: Our Index is equal to 1, and Lib Portion Index is equal to 0.8.

$$O = \frac{|A \cap B|}{|B|} = \frac{|\{2, 3, 4\}|}{|\{2, 3, 4\}|} = \frac{3}{3} = 1$$

$$LP = \frac{|A \widetilde{\cap} B|}{|B|} = \frac{|[2, 2, 3, 4]|}{|[2, 2, 3, 4, 4]|} = \frac{4}{5} = 0.8$$

Both of these metrics have their own advantages and disadvantages. For example, the Jaccard based metrics are able to find partial matches better than the Lib Portion based metrics. However, the Lib Portion Index works better than the Jaccard metrics when we are trying to identify the library in the bundle of libraries. In the case when we are searching for one target library in the bundle, Jaccard based metrics will have lower values for all target libraries, inversely proportional to the bundle size. However, the Lib Portion based metrics will not be affected by the unknown bundle size. Only the size of the known target library will be the deciding factor for the resulting value.

Additionally, both known and unknown signatures might be empty lists. It means, that in certain situations the above mentioned metrics might end up dividing the numerator by zero. We defined few exceptions for the above mentioned metrics that handle the division by zero case. The exceptions are presented in Table 4.2.

For the case of the Jaccard based metrics, if both input lists are empty, then the intersection and the union of these two lists will also be empty, which implies that the numerator and the denominator are both equal to zero. For this case, we define the

Table 4.2: Similarity Metric Exceptions.

Given the unknown list U and the known list K,
the similarity value v is set according to the table.

Metric	Case	Exception
Jaccard / Jaccard Like	$\|U\| = 0$ and $\|K\| = 0$	$v \leftarrow 1$
Our / Lib Portion	$\|U\| = 0$ and $\|K\| = 0$	$v \leftarrow 1$
	$\|U\| > 0$ and $\|K\| = 0$	$v \leftarrow 0$

similarity metric to be 1, since we treat two empty lists as exactly matching lists. In all other cases, the similarity value will be defined, as the denominator will not be equal to zero.

For the case of the Lib Portion based metrics, similarly to the Jaccard based metrics, we define that if two empty lists are passed, then we treat them as exactly matching lists. However, if the unknown list has some items, but the known list is empty, we define the similarity value as 0. These two rules define that, for the Lib Portion based metrics, only empty lists are the same, and any non-empty list is not the same as the empty list.

An additional property of the modified metrics for lists that were defined above is that in the case when lists of unique elements (*i.e.*, a set) are passed, these two metrics behave exactly in the same way as the versions created for sets. This allows us to use those two metrics interchangeably, when we are sure that the input list only contains unique elements. Also, because of this reason, all of the matching methods described below only use the modified metrics.

As mentioned above in Section 4.2.1, the extracted signature of a function contains the name of the function, the list of statement tokens extracted from each statement defined in the function as well as the list of statement types of all statements defined in the function. In our framework we defined ten various methods of comparing two signatures based on the extracted data, which are described in a greater detail below.

- `fn-st-toks-v6`, `fn-st-toks-v5`, `fn-st-toks-v4`, `fn-st-toks-v3`, `fn-st-toks-v2`, `fn-st-toks-v1` — Variations of the method that uses lists of statement tokens.

  Our first attempt is the `fn-st-toks-v1` method. We take the signature for one function from the entire unknown signature. We compare the selected unknown function signature against all known function signatures. To compare two individual

function signatures, we take the list of extracted tokens from the unknown function and compare it against the list of tokens from the known function, using the Jaccard Like Index between the two lists of tokens. We then rank those similarity values between functions to find the most likely candidate function from the known signature list. From the ranking, we select the top matching function and save the name of the matched function from the known signature into the list of possible function names. However, if the top matching function in the created list of rankings has the similarity metric of 0, we add the dummy function name `__unmatched__` into the list of possible function names. After attempting to match every function from the unknown list, we end up with the list of either dummy function names `__unmatched__` or the names of functions from the known list. To produce the final similarity metric between two signatures, we match this resulting list with the list of function names from the known signature using the above mentioned Jaccard Like similarity metric.

The method `fn-st-toks-v2` is a modification of the `fn-st-toks-v1` method. Since the previous method first picks the function from the unknown signature to match against functions from the known signature, it means that the previous method forces the unknown function to be matched against some known function. This gives some advantage to functions that are ordered first in the unknown signature. To counteract this problem, the updated method picks one function from the known signature and tries to match it against all functions in the unknown signature. In this case, if the known function is present in the unknown signature, then it will get matched. However, if it is not present, it may be matched partially to some other function since the known function is being 'forced' to get matched to some function. The additional change in this method is that we are no longer storing the names of possible functions from the known list, but rather we are storing the index mapping between the unknown list and the known list. This way eliminates the need to be aware of the repeated function names, or using a real and existing function name as a dummy name by accident.

The method `fn-st-toks-v3` tries to counter the problem of forcing either a function from the unknown signature or a function from the known signature to be matched. In this method, all functions from the known and unknown lists are pair-wise compared using Jaccard Like Index and ranked. To create the index for ranking, we use the list of statement tokens from each unknown and known function and compare those lists directly using Jaccard Like Index function. After pairing and ranking, matched pairs are removed according to the ranked value, so that the top matching pair is removed first. After matching pairs are removed, all subsequent pairs, that use already encountered functions, are skipped, since either one or two of the participat-

ing functions in a pair would already be matched in a better match. Unfortunately, this method turned out to be very performance intensive, and instead of pursuing this approach we attempted to make improvements to the `fn-st-toks-v2` method.

The method `fn-st-toks-v4` is based on the `fn-st-toks-v2` method. This method attempts to take into account the information about the similarity value between two matched functions when calculating the final similarity score between two signatures. For example, in all of the previous methods, we would match as many functions as possible between the unknown and known signatures and then create the final similarity metric based on the created lists. However, this would not account for the similarity value of each individual matched pair between the unknown and known function. We create this weighing factor as a summation of all similarity values between all matched functions over the number of those matched functions. This means that if all functions got matched with 100% similarity, the weighing factor will be 1. However, the factor will be less than 1, if some of the functions were matched with less than 100% accuracy. When constructing the final similarity value between two signatures, we calculate the Jaccard Like Index in the same way as method `fn-st-toks-v2` would and multiply the resulting value by the weighing factor obtained earlier.

The method `fn-st-toks-v5` is based on the `fn-st-toks-v2` method. In all of the methods above, we are trying to match as many functions as possible, producing the mapping between functions with various accuracy values. Some functions would get matched with 100% accuracy, meaning all tokens in both functions were exactly the same. However, there are some function matches that have less than 100% accuracy, meaning we would be only partially sure that they are similar. We observed that some functions, which matched with less than 100% accuracy, would increase the final similarity metric to the point where the top match of some unknown library would be a completely different library from what it should have been. The modification introduced in this method is that for the final similarity computation, we only keep functions from the unknown signature that got matched to functions from the known signature with 100% accuracy value. This method eliminates uncertain matches that could significantly increase the final similarity value.

The method `fn-st-toks-v6` is based on the `fn-st-toks-v5` method. In this method we also keep only functions that got matched with exactly 100% accuracy. The difference from the `fn-st-toks-v5` method is that for the final similarity value we are not using the Jaccard Like Index, but rather the Lib Portion Index. In short, this method finds as many precisely matched functions as possible and ranks target libraries based on which proportion of the library got matched.

- `fn-names-our`, `fn-names-jaccard` — Two variations of the method that uses function names.

  Both of these methods only use the names of functions from the unknown and known signatures to match them. Furthermore, these two methods only use unique names among signatures (*i.e.*, the set of names) when comparing two signatures. The difference between these two methods is that the method `fn-names-our` uses the Lib Portion Index for the final value, while the method `fn-names-jaccard` uses the Jaccard Like metric for the final value.

  These two methods are very simple and quick to compute. These methods work very well when there are lots of named functions with distinct names. However, in JS anonymous functions are commonly used, which do not have any name. In order to still keep track of anonymous functions, we encoded their names with the `[anonymous]` keyword as well as their depth level relative to other functions. For example, there may be a top level anonymous function with the complete given name as `[anonymous]`, which has other functions inside of it. For example, it might have one named function and two anonymous functions, with complete given names as `[anonymous]:>>:a` for named function, and `[anonymous]:>>:[anonymous]` for both anonymous functions. In total, in this example, there are four functions defined. However, these two methods will only be able to "see" three functions, as two anonymous functions will have the same complete name. Since there might be lots of anonymous functions in real code, these two methods will miss "sibling" anonymous functions — those defined at the same depth level.

- `fn-names-st-toks` — One method that matches based on the combination of function names and lists of statement tokens.

  This method attempts to combine approaches employed by the method `fn-names-jaccard` and by the method `fn-st-toks-v1`. While we can see that for named functions, the method `fn-names-jaccard` works, it does not work well for anonymous functions. Therefore, in this method we are trying to use the approach of `fn-st-toks-v1` when matching anonymous functions. This method splits incoming unknown and known signatures into two parts. One part are anonymous functions, which are defined as functions that have the last part in their complete given name equal to `[anonymous]`. The second part are all other functions, as there should be only named functions left at this point. After partitioning function signatures into those groups, we compare matching groups separately. For all anonymous functions in the unknown and known signatures we are employing the method exactly as it is described in `fn-st-toks-v1`. For all named functions we are employing the

method exactly as it is described in `fn-names-jaccard`. After we have two distinct mappings, one between anonymous functions and one between named functions, we combine those two mappings to calculate the resulting similarity metric between two signatures using the Jaccard Like Index.

- `fn-st-types` — One method that uses statement types.

  This method is very similar to the `fn-st-toks-v1` method. It matches every function from the unknown signature to functions from the known signature using the Jaccard Like Index and then uses the Jaccard Like Index to produce the resulting similarity metric between two signatures. The only difference between this method and the `fn-st-toks-v1` method is that this method uses the list of types of each statement in the function instead of the list of tokens extracted from statements of the function.

All of the methods listed above produce the output in the same format. This allows us to use any one of those methods to perform analysis between the signature of the application file and the signature of the library file. We will evaluate the performance of all of the methods and select the best performing method.

4.2.5 Stage 3: Compare the Application File Signature Against All Candidate Library Signatures

During the previous steps we already completed the creation of signatures for all library files and application files, as well as the creation of the list of candidate libraries for each application file. In this step, we will compare the signature of one application file against the signatures of all versions of all libraries mentioned in the candidates file. This algorithm is created to address the issue of libraries being bundled together into one file. It takes a list of multiple candidate libraries and evaluates how likely each of those candidates is to appear in the bundle file. We can assume that the simple case of a single library in a file is a special case of a bundle file, when we have only one library in our bundle file. Since this algorithm works in a generic way with the list of candidates of any size, this method will also work for files that only have one candidate.

Algorithm 2 presents the approach used to perform this step. This function relies on the existence of functions implementing the $comp\_type\_fn$ interface, which is described in greater detail in Section 4.2.4. The similarity returned by the matching method is used in Algorithm 3 to find which version of the particular library is more likely to be present in the file, compared to other versions of the same library. The algorithm reorders all versions of

Algorithm 2 Bundle Similarity Function

```
1: function BUNDLE_SIMILARITY(sig, cand)
2:     later ← [ ]
3:     exact_matches ← [ ]
4:     partial_matches ← [ ]
5:     sCand ← SortByMostLikely(cand)              ▷ Place most likely candidate first

6:     for all c ∈ sCand do                        ▷ Find exact matches among all candidates
7:         top_match ← COMPARE(sig, c)
8:         if top_match.similarity = 1 then
9:             exact_matches ← exact_matches + top_match
10:            sig ← {x|x ∈ sig and x ∉ top_match.mapping}      ▷ Remove matched fns
11:        else
12:            later ← later + c
13:        end if
14:    end for

15:    for all c ∈ later do                        ▷ Find partial matches among remaining candidates
16:        top_match ← COMPARE(sig, c)
17:        partial_matches ← partial_matches + top_match
18:        sig ← {x|x ∈ sig and x ∉ top_match.mapping}
19:    end for

20:    remaining_functions ← sig
21:    return (exact_matches, partial_matches, remaining_functions)
22: end function
```

Algorithm 3 Compare Function

```
1: function COMPARE(sig, candidate)
2:     matches ← [ ]
3:     versions ← LoadAllVersions(candidate)        ▷ Load signatures of all versions
4:     reordered ← ReorderVersions(versions)        ▷ Place newer versions earlier

5:     for all v ∈ reordered do
6:         matches ← matches + comp_type_fn(sig, v)   ▷ One of comparison functions
7:     end for

8:     sorted_matches ← SortByMostLikely(matches)
9:     top_match ← sorted_matches[0]
10:    return top_match
11: end function
```

a candidate library as shown on Line 4. The function $ReorderVersions$ on Line 4 shuffles versions such that newer versions are placed roughly before older versions. In detail, this function takes the ordered list of versions, picks version which is near the end of the list, and from the selected version it steps backwards and forwards, until it covers all versions in the list. For example, given the list of versions $[1, 2, 3, 4, 5, 6]$, the function will order the list as $[5, 4, 6, 3, 2, 1]$. This behaviour allows us to check versions that are newest before older versions, and allows us to slightly optimize Algorithm 3. The algorithm uses one of the matching methods to compare an unknown signature to signatures of all versions of the library and ranks those versions based on the similarity value. It then returns the top version match to Algorithm 2.

The value returned from Algorithm 3 is used as a top match for further steps. If the top match has the 100% similarity value, then this version and this candidate is saved as the exact match, otherwise, it is saved for later, for the second iteration of the search. The mapping between matched functions of signatures is used to indicate which functions of the unknown signature got assigned to the particular version of a particular library. That is required, so that when we are matching against the next candidate, we will not match the same function twice, rendering the results incorrect. The algorithm will proceed to finish the first pass through all candidates, and after it is done, all candidates for the file that got ranked with 100% similarity would be found and all functions that belong to those libraries would be mapped. In the second iteration, the algorithm does a similar matching with the remaining candidates and remaining functions. It will use Algorithm 3 to compare the remaining functions to all versions of a candidate and return the highest matching version.

However, it stores matches as-is, without picking only those that have 100% similarity value. This second iteration is attempting to find as many libraries as possible, even if we cannot detect them with exact certainty. This allows us to detect libraries that were modified significantly with the minifier, or detect libraries for which we do not have the exact version available.

Chapter 5

Experiments and Discussion

This section provides the results of various experiments as well as discussions of the experiments.

5.1 Results for Selected Applications

In this section we present the results of the algorithm analysis for those applications that we manually verified. In the manual analysis, we had 17 instances of library name+version pairs across eight different applications. The application names and versions that were used for manual verification are shown in Table 5.1. The results of the manual verification using all various signature comparison methods are presented in Tables 5.2 and 5.3. Next, we present some issues with the matching methods that we observed, and discuss what might have been a cause of those issues.

In both of the tables showing results, the column 'N' shows the detection of library name and the column 'NV' shows the detection of library name and version. The value 'E' shows the exact match, the value 'P' shows the partial match, the value 'x' shows no match detected. Additionally, there are two rows showing the total count for each method. One of the two counts displays the number of total vulnerable libraries that should be found across all applications and files. The second one of the two counts displays the number of total libraries that each particular method has detected. From these two total counters we can report the precision and the recall for all methods, which are presented in Table 5.4. The table shows the precision and recall of all methods considering only exact matches reported by methods. We have manually verified that all results that were returned by all methods

Table 5.1: Applications of interest.

ID	Application Name	Version
1	br.com.williarts.radiovox	20008
2	com.atv.freeanemia	1
3	com.paynopain.easyGOband	18
4	com.tiny.m91392d54e89b48a6b2ecf1306f88ebbb	300000016
5	com.tomatopie.stickermix8	10
6	io.shirkan.RavKav	1800000
7	net.jp.apps.noboruhirohara.yakei	102008
8	it.wemakeawesomesh.skitracker	102081

are either non-guesses or false-positives. We have not found a single match returned by all methods that we considered a guess during the initial manual analysis of eight applications mentioned in Table 5.1. We also found that the single false positive of the `fn-st-toks-v6` method is the `cyber-js@1.0.3` library, which we discuss further in Section 5.2.1. In short, if we have source data of higher quality as well as if we perform better filtering for the vulnerable set of libraries, the precision of `fn-st-toks-v6` will increase.

If we consider only the results for exact matches, we can see that the two matching methods `fn-st-toks-v6` and `fn-names-our` perform the best in detecting the names of libraries. However, the method `fn-names-our` cannot detect exact versions of libraries as precisely as the `fn-st-toks-v6` method can. For example, the usage of `jquery@2.2.2` from App 4 and the usage of `moment@2.8.3` from App 8 are detected by `fn-st-toks-v6`, but those are not detected by the `fn-names-our` method.

In the case if we include exact and partial matches, none of the methods can detect usages of `lodash@2.4.2` in App 4 and `jquery-ui@1.11.1` in App 6. Additionally in this case, the method `fn-st-types` is unable to detect the usage of the `lodash@2.2.2` library in App 4. However, all other methods can detect the name of the library that is being used in all other cases. If we attempt to search for the name and version of the library, then not all methods perform as well. The best performing method, `fn-st-toks-v6`, could not detect two of the usage instances out of the maximum amount it could detect. In particular, this method does not detect the library `lodash@2.4.2` from App 4 and the library `moment@2.17.1` from App 1. There are two other methods that match the performance of this method — `fn-names-st-toks` and `fn-st-toks-v5`. The second best method in detecting names of libraries, `fn-names-our`, performs as one of the worst metrics — there is one rivaling this metric (`fn-st-toks-v3`), and only one that is worse

Table 5.2: Detection of vulnerable libraries by the algorithm.

detecting: N — the name, NV — the name+version.
match: E — exact, P — partial, x — none.

ID		Library@Version	*fn-st-toks-v6*		*fn-names-our*		*fn-st-types*		*fn-names-st-toks*		*fn-names-jaccard*	
			N	NV	N	NV	N	NV	N	NV	N	NV
1		`angular@1.5.3`	E	E	E	E	P	x	P	P	P	P
		`moment@2.17.1`	P	x	P	x	E	E	P	x	P	x
2		`jquery@1.11.1`	E	E	E	E	E	E	E	E	E	E
3		`angular@1.4.3`	E	E	E	E	P	x	P	P	P	P
		`jquery@2.1.4`	E	E	E	E	E	E	E	E	E	E
		`moment@2.9.0`	P	P	P	x	P	P	P	P	P	P
4		`angular@1.4.3`	E	E	E	E	P	x	P	P	P	P
		`lodash@2.4.2`	x	x	x	x	x	x	x	x	x	x
		`jquery@2.2.2`	E	E	E	x	x	x	P	P	P	x
5		`jquery@1.11.2`	E	E	E	E	E	x	E	E	E	E
		`bootstrap@3.3.2`	E	E	E	E	E	E	E	E	E	E
6		`jquery@2.1.1`	E	E	E	E	E	E	E	E	E	E
	*	`jquery-ui@1.11.1`	x	x	x	x	x	x	x	x	x	x
7	*	`jquery@1.9.0`	P	x	P	x	P	x	P	x	P	x
8		`jquery@2.1.1`	E	E	E	E	E	E	E	E	E	E
		`angular@1.3.0`	E	E	E	E	E	E	E	E	E	E
		`moment@2.8.3`	E	E	E	x	E	E	E	E	E	E
		Exact	12	12	12	10	9	8	8	8	8	8
		Exact+Partial	15	13	15	10	14	9	15	13	15	12
Total vulnerable to detect			17	15	17	15	17	15	17	15	17	15
Total detected libraries			13		64		45		8		8	

Table 5.3: Detection of vulnerable libraries by the algorithm.

ID		Library@Version	*fn-st-toks-v5*		*fn-st-toks-v4*		*fn-st-toks-v3*		*fn-st-toks-v2*		*fn-st-toks-v1*	
			N	NV	N	NV	N	NV	N	NV	N	NV
1		`angular@1.5.3`	P	P	P	P	P	P	P	P	P	P
		`moment@2.17.1`	P	x	P	x	P	x	P	x	P	x
2		`jquery@1.11.1`	E	E	E	E	E	E	E	E	E	E
3		`angular@1.4.3`	P	P	P	P	P	x	P	x	P	x
		`jquery@2.1.4`	E	E	E	E	E	E	E	E	E	E
		`moment@2.9.0`	P	P	P	P	P	P	P	P	P	P
4		`angular@1.4.3`	P	P	P	P	P	x	P	x	P	x
		`lodash@2.4.2`	x	x	x	x	x	x	x	x	x	x
		`jquery@2.2.2`	P	P	P	x	P	x	P	P	P	P
5		`jquery@1.11.2`	E	E	E	E	E	E	E	E	E	E
		`bootstrap@3.3.2`	E	E	E	E	E	E	E	E	E	E
6		`jquery@2.1.1`	E	E	E	E	E	E	E	E	E	E
	*	`jquery-ui@1.11.1`	x	x	x	x	x	x	x	x	x	x
7	*	`jquery@1.9.0`	P	x	P	x	P	x	P	x	P	x
8		`jquery@2.1.1`	E	E	E	E	E	E	E	E	E	E
		`angular@1.3.0`	E	E	E	E	E	E	E	E	E	E
		`moment@2.8.3`	E	E	E	E	E	E	E	E	E	E
		Exact	8	8	8	8	8	8	8	8	8	8
		Exact+Partial	15	13	15	12	15	10	15	11	15	11
Total vulnerable to detect			17	15	17	15	17	15	17	15	17	15
Total detected libraries			8		8		8		8		8	

Table 5.4: Methods' precision and recall.

When detecting names or names and versions.

Method	Names		Names–Versions	
	Recall	Precision	Recall	Precision
`fn-st-toks-v6`	$\frac{12}{17}$ (71%)	$\frac{12}{13}$ (92%)	$\frac{12}{15}$ (80%)	$\frac{12}{13}$ (92%)
`fn-names-our`	$\frac{12}{17}$ (71%)	$\frac{12}{64}$ (19%)	$\frac{10}{15}$ (67%)	$\frac{10}{64}$ (16%)
`fn-st-types`	$\frac{9}{17}$ (53%)	$\frac{9}{45}$ (20%)	$\frac{8}{15}$ (53%)	$\frac{8}{45}$ (18%)
`fn-names-st-toks`	$\frac{8}{17}$ (47%)	$\frac{8}{8}$ (100%)	$\frac{8}{15}$ (53%)	$\frac{8}{8}$ (100%)
`fn-names-jaccard`	$\frac{8}{17}$ (47%)	$\frac{8}{8}$ (100%)	$\frac{8}{15}$ (53%)	$\frac{8}{8}$ (100%)
`fn-st-toks-v5`	$\frac{8}{17}$ (47%)	$\frac{8}{8}$ (100%)	$\frac{8}{15}$ (53%)	$\frac{8}{8}$ (100%)
`fn-st-toks-v4`	$\frac{8}{17}$ (47%)	$\frac{8}{8}$ (100%)	$\frac{8}{15}$ (53%)	$\frac{8}{8}$ (100%)
`fn-st-toks-v3`	$\frac{8}{17}$ (47%)	$\frac{8}{8}$ (100%)	$\frac{8}{15}$ (53%)	$\frac{8}{8}$ (100%)
`fn-st-toks-v2`	$\frac{8}{17}$ (47%)	$\frac{8}{8}$ (100%)	$\frac{8}{15}$ (53%)	$\frac{8}{8}$ (100%)
`fn-st-toks-v1`	$\frac{8}{17}$ (47%)	$\frac{8}{8}$ (100%)	$\frac{8}{15}$ (53%)	$\frac{8}{8}$ (100%)

than this method (fn-st-types).

There are few usage instances among 17 of a particular interest, which we will discuss in more detail.

1. First of all, even though `lodash@2.4.2` from App 4 is downloaded and present in our dataset of vulnerable libraries, it is not matched using any of the ten different signature comparison methods.

2. Second, the library `jquery-ui@1.11.1` from App 6 is not present in our dataset of vulnerable libraries. As a similar example, the library `jquery@1.9.0` from App 7 of this particular version is not present in our dataset, however there are other versions of this library. Unlike `jquery-ui@1.11.1`, the name of `jquery@1.9.0` is detected at least partially by some of the matching methods. Which raises questions as to why at least the name of `jquery-ui@1.11.1` cannot be detected by some of the methods.

3. Third, in the case of the library `moment@2.17.1` in App 1, only method fn-st-types is able to detect the name and version exactly, but other methods are only able to partially match the name of the library and are unable to find the version of the library.

4. Lastly, in the case of library `moment@2.8.3` in App 8, the only failed method is fn-names-our even though other methods are able to detect both name and version exactly.

5.1.1 The Case of `lodash@2.4.2` and `jquery-ui@1.11.1`

Investigating issues 1 and 2, we found that both of these libraries were not among the candidate libraries for files where they were used in their respective applications. This is a limitation of the candidate creation. When we are creating the list of candidates for each file in the application, there are two methods that are attempted. In the case of application files containing these two library+version usage instances, the creation of candidates fell back on using the second method. The second method limits the number of candidate libraries to ten distinct library names, and in the case of these two application files containing these two library usage instances, other libraries were estimated to be better candidates. Because of this, the actual matching using any method would not even attempt to match against `lodash` or `jquery-ui`.

5.1.2 The Case of `moment@2.17.1`

Investigating issue 3 of `moment@2.17.1` from App 1, we found that this file in the application was minified. However, this file contained all comments, despite all variable names being renamed into shorter ones. This is a very unusual behaviour of a minifier, so it leads us to believe that custom settings for a minifier were being used in this case. Additionally, the version of this library available on npm only distributes a non-minified variant of this version. Which means, in this case we are comparing an unknown signature of a minified file against the known signature of a non-minified file. This case is particularly difficult for all signature matching methods that we defined. Because of the minification, we can assume that names are all changed, sometimes the order of statements is changed, and sometimes the statements themselves are completely changed.

Because of the minification, all methods that rely on names of functions are highly likely to fail. Such methods are `fn-names-our`, `fn-names-st-toks` and `fn-names-jaccard`.

Additionally, we have a lot of methods that rely on statement tokens, such as `fn-st-toks-v6`, `fn-st-toks-v5`, `fn-st-toks-v4`, `fn-st-toks-v3`, `fn-st-toks-v2`, `fn-st-toks-v1` and `fn-names-st-toks`. These methods use the list of tokens of the function to perform the comparison and are prone to mistakes. The token based signature of the function captures a lot of information about the statements and tokens being used in the function. We attempt to ignore some statements that we believe do not bring enough information about the function and also impact the matching when these tokens are modified by a minifier. For example, we exclude variable declarations from statements that we capture. However, there might still be cases where a minifier greatly modifies the function and these methods might fail.

The last method that we define (`fn-st-types`) relies on the type of each statement in the function. This method may potentially extract very little information about each particular function. For example, in JS, statements are constructed out of expressions, and there is one particular expression type that allows nesting as many other expressions as needed. This means, that the entire function can be done as one statement, even if it will have a lot of expressions performing tasks. In this case, this method might not be able to adequately capture enough information about a function. Additionally, some minifiers reorganize code, *e.g.*, switch between a for loop and a while loop based on which will end up being shorter. In those cases, this method might not work very well. However, this method will be able to process files that were minified by only renaming the variables.

5.1.3 The Case of `moment@2.8.3`

Investigating issue 4 of `moment@2.8.3` from App 8, we found that the top match was the library `moment@2.8.1`, *i.e.*, an earlier version of the same library. The detected match was the exact match, *i.e.*, 100% of the library of version `2.8.1`, even though there were some functions from the application file that were left unmatched. This erroneous match comes from the combination of specifics of the `fn-names-our` matching method, as well as Algorithm 3. As mentioned in the details of this matching method (in Section 4.2.4), it uses the set of names of found methods to compare against the library. It turned out that the set of names in the version `2.8.3` is a superset of names in the version `2.8.1`. In other words, all function names from the version `2.8.1` were not renamed in the version `2.8.3`. Only new functions were added in the version `2.8.3`. Which means that the metric of this matching method produces 100% match for both `2.8.1` and `2.8.3`, since it is only evaluating the proportion of the library set that was used in the unknown set. Additionally, the created signature for the file in the application is exactly the same as the signature created for the `moment@2.8.3` library, and the created signature has 231 functions detected. However, the produced value of similarity by the `fn-names-our` method is 205/205 when compared to the same signature of `moment@2.8.3`, instead of 231/231. This shows that during the creation of the signature, there were few functions that ended up having the same name. For example, there could be two different anonymous functions in two different statements of the same function. In this case, the name created for both functions will be the same, even though their source code, and hence tokens, is different. This method is not only unable to capture the differences between functions, but also it is unable to detect multiple functions with the same name.

The second part comes from the optimization of Algorithm 3. As mentioned, the chosen matching method finds more than one library version with 100% match. Algorithm 3, comparing an unknown signature to all versions of the library, assumes that there is only one exact match among all versions of the library, for optimization purposes. Additionally, we are assuming that the majority of applications will use somewhat new versions of libraries as dependencies. Because of this, we are reordering the list of versions, so that newer versions are checked before checking older versions. This allows us to find exact matches slightly faster. However, as soon as the first version with 100% match is found, Algorithm 3 terminates its search and returns that version as the exact match. In the case of this usage instance following the steps outlined in Section 4.2.5, versions were ordered in such a way that the version `2.8.1` appeared earlier than `2.8.3` and therefore was reported as the exact match, according to reasons described above.

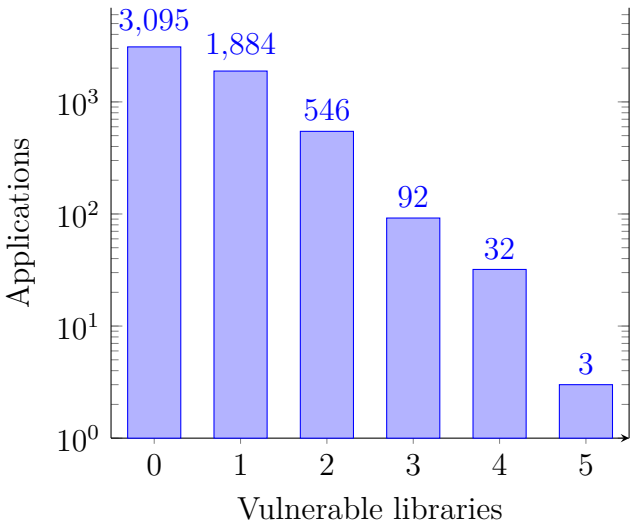

Figure 5.1: Number of vulnerable libraries per application.

5.2 Results for the Entire Dataset

In this section we show the results of the algorithm on the entire corpus of 5652 Android applications scraped from the Google Play Store.

As we can see from Figure 5.1, the majority of applications have no detected vulnerable libraries. However, there are still 2557 or 45.24% of applications in our corpus that have at least one vulnerable library. Additionally, the highest number of detected vulnerable libraries per application is five, which were detected only among three applications. Those three applications with the highest number of vulnerable libraries are shown in Table 5.5. We do not report the application IDs, since we do not wish to disclose those applications.

While manually analysing these applications, we saw that all three applications: 1. are developed by different developers; 2. have newer versions available on the Google Play Store; 3. and have the detected libraries in different files. According to our tool, each of the applications include different versions of the same library twice. For example, the first application is including two different versions of the `angular` library — `angular@1.4.3` and `angular@1.4.5`. We manually verified all of the reported libraries and found that the first application indeed uses `lodash@1.3.1`, `jquery@2.1.4`, and two different versions of `angular`. When we checked the file containing the library `cyber-js@1.0.3` according to our tool, we found that the file did not contain this library. When we verified the second application, we found that it indeed uses all of the library names and versions that

Table 5.5: Applications with five vulnerable libraries.

LAV – Last App Version, RY – Release Year

ID	LAV	RY	Library@Version

1	No	2017	`angular@1.4.3`
			`angular@1.4.5`
			`cyber-js@1.0.3`
			`jquery@2.1.4`
			`lodash@1.3.1`

2	No	2017	`angular@1.4.3`
			`jquery@1.11.3`
			`jquery@2.1.4`
			`lodash@4.17.2`
			`moment@2.10.6`

3	No	2017	`angular@1.2.28`
			`angular@1.3.13`
			`bootstrap@3.3.2`
			`jquery@1.11.0`
			`moment@2.10.6`

were detected by the tool. Which means that this library also used the same library of two different versions — `jquery@1.11.3` and `jquery@2.1.4`. When we verified the third application, we found that it also uses two different versions of `angular` — `1.2.28` and `1.3.13`. We also found that all of the remaining reported libraries were present in the third application, with the exact versions reported by the tool.

We also investigated what are the most used libraries among the ones we detected. Figure 5.2 shows the list of all detected library names among applications in our corpus. We can see that there are a lot of very popular packages, as well as three less popular packages — `cyber-js`, `zhangranbigman`, and `chromedriver126`. Figure 5.3 shows the top 15 of the most detected library name and version combinations among the applications. We can see that it is dominated by various versions of different popular packages. However, there is one of the less popular packages appearing on the list — `cyber-js@1.0.3`. From both figures, we can see that two of the most popular libraries — `jquery` and `angular` — are dominating the charts.

As mentioned above, there are a few less popular packages that were detected. We will look at all of them in greater detail.

5.2.1 The Case of `cyber-js`

The library `cyber-js` [30] was detected to be used in 152 distinct applications according to Figure 5.2. Additionally, the specific version `cyber-js@1.0.3` was also detected in 152 distinct applications, according to Figure 5.3. This means that regardless of the number of various versions that this package has published to npm, only one version was matched. Investigating the given version, we found that the library only has one file, which is also the main file, presented in Listing 8.

This file only has one function. In the function, the string `"Hello"` is printed on the console. This function is equivalent to the sample `"Hello world!"` program in JS. Additionally, this package does not contain any scripts in the metadata file `package.json` that would be executed during the installation. This shows that the package cannot be vulnerable, since it is not executing anything during the installation or accepting any parameters in the only function this library exports. However, according to the Snyk report [15] for this library, all versions of the given library are affected with the "Directory Traversal" type of vulnerability. While manually investigating all of the versions of this library, we found that only in versions `1.0.6` and `1.0.7` this library has more code in few separate files beside the simple `"Hello world!"` program. In both of those versions, the library performs access to the file-system, using one of the built-in Node.js modules, that

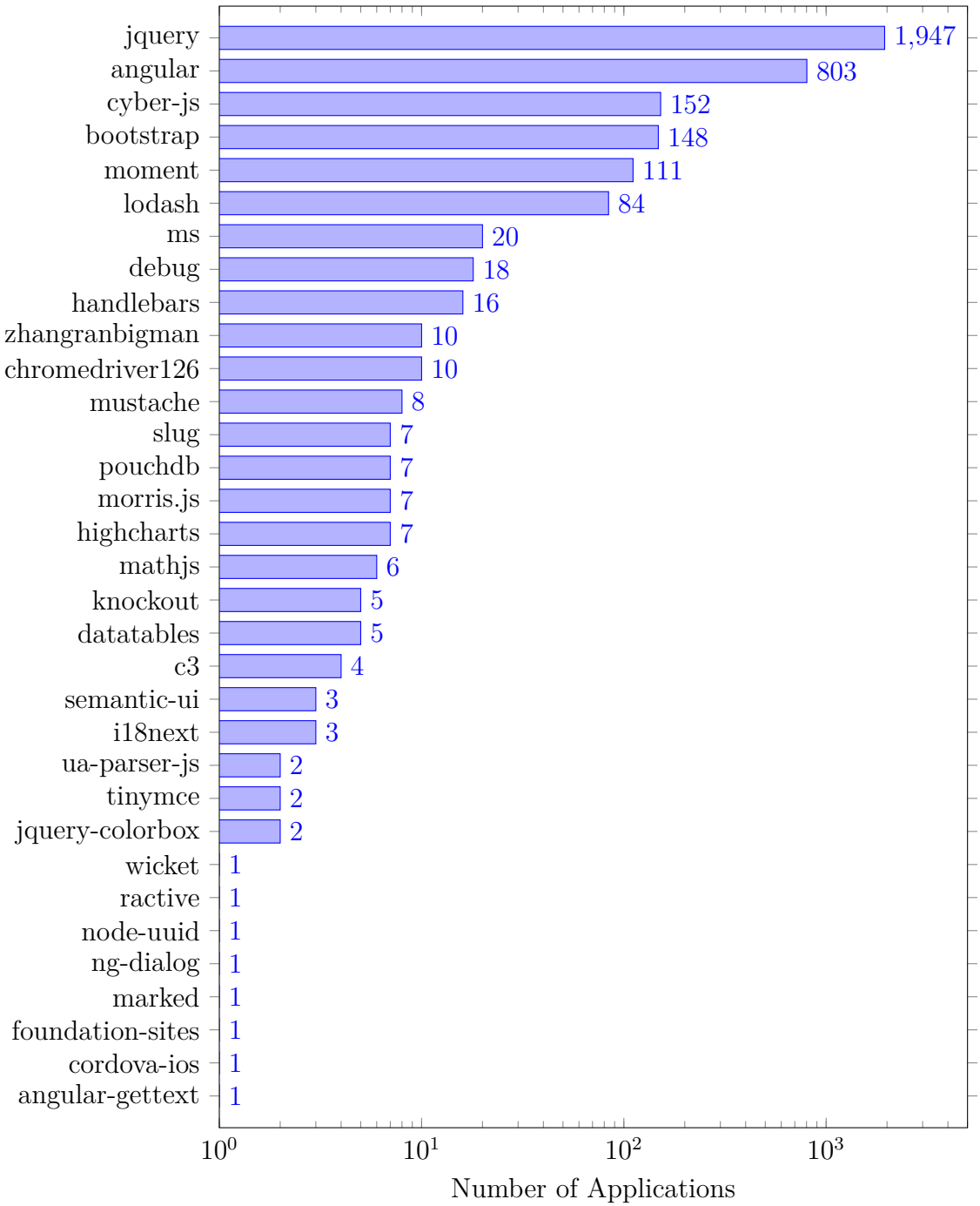

Figure 5.2: All detected library names.

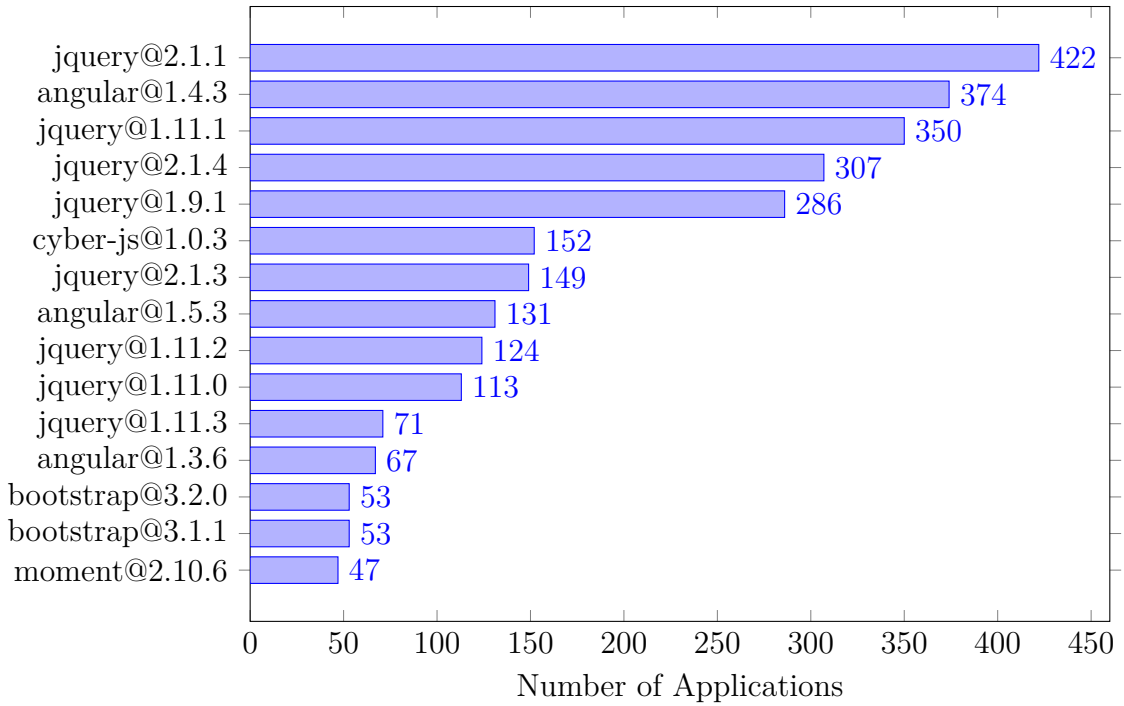

Figure 5.3: Top 15 most detected library versions.

```
/**
* Author: Alex Munoz
* Cyber JS
*/

exports.printMsg = function() {
  console.log("Hello");
}
```

Listing 8: Contents of `cyber-js@1.0.3` main file.

leads us to believe that these versions expose a vulnerable "Directory Traversal". Which means that the vulnerability report for this library should only have included versions starting (and including) `1.0.6`.

Additionally, this package only contains one function with only one very common function call — to print a string to console. This case poses difficulty for some of our matching methods. For example, the method `fn-st-toks-v6` uses the portion of the matched library as a final similarity metric between two files. In this case, if the unknown target file contains one function anywhere in the file that only prints a string to console, the method `fn-st-toks-v6` will report that the unknown file exactly matches to this library. Some methods that use the Jaccard Index as their final matching metric might be able to counter such cases. That is because the Jaccard Index is comparing the size of common parts between the unknown file and the known file relative to sizes of both files. For example, many libraries will have more than one function defined in the code. This will reduce the value of the Jaccard Index greatly, as the size of the library signature increases. Unfortunately, the metric based on the Jaccard Index will favour larger libraries when attempting to match unknown files. Additionally, the Jaccard Index will have questionable behavior when attempting to detect libraries in the bundle files. Since at every iteration of the matching loop we are removing those functions that got matched to some target library during the previous iteration, it means that the size of the unknown file signature is shrinking, which will affect the value of the Jaccard Index. This means that the similarity value for two libraries in the bundle will depend on the order of matching libraries. Additionally, the similarity value of two different detected libraries in the bundle will not be comparable, since those values will have different semantic. For the first library, the semantic is *"how close is the unknown bundle to the first given known library"*. However, for the second library, the semantic is *"how close is the unknown bundle without the first library to the second given known library"*.

5.2.2 The Case of `zhangranbigman`

The library `zhangranbigman` only has one version published on npm [31]. According to the Snyk report [16], this library has the "Directory Traversal" type of vulnerability affecting all versions of this library.

However, after manually verifying the source code of the library, we found that this library was not created for the client side. This comes from the fact that this library accesses internal Node.js modules, such as `http`, `fs`, and `url`, which are not meant to be used on the client side.

Additionally, this library is distributed as multiple files. However, our approach relies on the fact that a client side library is distributed as one file with its full functionality. So, in the case of this library, we cannot detect functions in other files, but we can only detect functions in the main file of the library.

5.2.3 The Case of `chromedriver126`

The library `chromedriver126` [29] appears to be the fork of an older version of the more popular library `chromedriver` [28]. According to the Snyk report [6], this library has the "Resources Downloaded over Insecure Protocol" type of vulnerability affecting all versions of this library.

From the manual verification, we find that `chromedriver` is a tool to programmatically control the Chrome browser for testing purposes. Because of this, this library should have been filtered as one that could not appear on the client side. We performed some blacklisting of libraries from the list of top 100 most used libraries on npm. However, this library is not as popular and therefore it was not filtered out.

5.2.4 Summary of the Results for the Entire Dataset

From the results for the entire dataset, we can see that the majority of detected libraries are various versions of very popular libraries — `jquery`, `angular`, `bootstrap`, and so on. However, we also observed few cases of server-side libraries or tools which were not removed from the reference dataset and hence our tool matched those libraries. We also saw that one library that we manually investigated was detected as a false positive — `cyber-js`. This library does not appear in actual applications, however the tool reported the presence of this library. Fortunately, we know exactly why it was detected, so we can come up with some possible mitigations against cases like this.

5.3 Threats to Validity

We identified a few threats to the validity of our results. These threats are primarily related to how we construct the signature for the list of candidates as well as the list of candidates itself. We also report on some threats related to the collection of the reference libraries.

5.3.1 Construction of the Candidate List

One of the threats to the validity for our results might be the fact that we are limiting the list of library candidates for each file to ten. We selected this arbitrary value for performance reasons. Namely, in our algorithm we are comparing each signature from all files of all applications against each signature of all files of all libraries that are selected as candidates. In case we selected to not limit the candidate list, our algorithm would take too much time to compare one signature against all signatures of approximately 600 libraries in our set. So, we selected the rather low value of ten candidates for each application file signature. However, some unknown files among applications might be very large bundles, containing more than ten distinct libraries. Because of this, our method might not be able to find all of the vulnerable libraries in the application, but it will find some of them.

In order to identify candidates for an unknown file in the application, we are comparing literals of the library and literals of the unknown file. In our case, literals are string values, number values, and regular expression values. The calculation of a candidate is described in detail in Section 4.2.3. The index is calculated between a set of literals created from the unknown file and a set of literals created from all versions of the library. We perform a simple filtering of the most common literal values, such as empty strings and integer numbers in the interval $[-1, 5]$. However, even with filtering, we saw a large number of literal values that do not bring any utility. For example for string literals, single character values are very common, such as a single space character, a new line character, a tab character, a quote, a parenthesis and others. We also saw string literals with the value of an integer number. All of these literal values are extremely common and are seen between different libraries. It means that because of these literals, we might mark more libraries as candidates than we should. Combined with the fact that the candidate list is limited to only ten libraries, we might end up in the situation where the library that should be compared against (*i.e.* the real candidate), is ranked lower than some other library to the point where the real candidate library is no longer on the list of candidates. Because of this issue, it might be the case that for some unknown files we have not created proper candidates and missed some libraries that were present in the unknown file. However, even if some libraries were incorrectly added into the candidate list, they would be going through the more precise matching and will get eliminated from results.

5.3.2 Gathering the Reference Library Dataset

Additionally, the source of reference libraries could be a reason for the algorithm to miss some library matches. That is because originally, npm was a repository of packages for the

Node.js platform. However, since npm has very little restrictions on what can be uploaded, more web related platforms started using it. For example, many front-end libraries started to be distributed on npm such as jquery, underscore and others, packages for Cordova are distributed via npm, as well as packages for some languages that compile into JS. Additionally, the source of vulnerability data that we are using — Snyk — is also not making any distinctions for packages that are distributed on npm. It simply stores the discovered vulnerability for the package, regardless of the fact that the vulnerability can only be present on the Node.js or in the browser. For example, the "Directory Traversal" type of vulnerability cannot be present in a browser, because a browser does not provide any file system access.

Because of these specifics, we cannot distinguish whether some vulnerable library was intended to be used on the server side or on the client side. However, the distinction has to be made, because the code written for the Node.js platform makes assumptions that do not hold true for a browser. For example, Node.js provides some internal modules, such as `path`, `fs`, and others, which are not provided in the browser. Additionally, libraries written for Node.js have access to functionality that allows to load the source code from other files. Which allows Node.js developers to ship libraries written across several files. However, before the language revision called `ES2015` was released, browsers did not have such a capability. The feature to load other files in JS was implemented by most of the browsers only recently.

For compatibility reasons, most libraries are still being distributed as one file containing all of the functionality, so that developers using the library and creating web sites for older browsers can simply include one library file and be able to use it that way. Because many library developers attempt to create libraries that are compatible with a variety of technologies — both old and new — we are able to greatly simplify our framework. Relying on this fact, we are able to simply find one main file containing the entire functionality of the library, analyse the file and use it as a reference. However, our method stumbles when it encounters a library written for the Node.js platform. Since we cannot make a distinction between two library types, our method will only analyse the main file of the library for the Node.js platform, that is most often simply the loader file of other files with actual functionality of the library.

Because we cannot detect sever-side libraries automatically, we should exclude them manually. We attempted to exclude non client-side libraries using a list of top 100 most dependent upon packages in npm as seen in Section 4.1.1. Since we only looked at the top 100, there are more libraries that did not get filtered out and that made it into our reference list.

Since some of the server side libraries are appearing in our reference list, we may compare unknown application files against some server side libraries or some development tools. Because of this, we may report on the existence of some server side libraries, which should not have been matched against.

There might be another reason why we miss some library matches. In the beginning npm allowed developers of libraries to unpublish versions of libraries from the repository. However, after some controversy related to the library called `left-pad` [46, 47], npm changed the rules regarding the removal of libraries and versions of libraries. Even though the rules are changed and it is more difficult to remove a certain version of a library from the registry, some older versions of libraries, that were removed before the change of rules are not restored and are not present in the registry. This means that some older applications might have been built with an old version of the library, which got unpublished from the registry, and therefore does not appear in our library reference list. Because of this, we might miss some exact matches when we are trying to search for certain libraries. Even though these libraries might show up as partial matches reported by our tool, since we primarily focus on the exact matches, those partial matches will be missed.

Chapter 6

Conclusions

6.1 Contributions

The first contribution in our work is that we come up with the method of creating a static signature for an arbitrary JS file. Such a signature represents the source code. The signature consists of two parts. The first part is created from hard-coded literals throughout the source code. The second part is created from all functions found in the file. For each function, we are extracting the name of the function, the list of tokens from each statement of the function and a list of types from each statement of the function. Additionally, extracted tokens only include tokens bringing useful information, while omitting some tokens that could interfere with matching, when we are dealing with the case of minified library.

The second contribution is our method of comparing two signatures between each other. Our comparison method produces a numerical value in the range $[0, 1]$ in order for us to be able to rank two comparison values between each other. Additionally, our comparison method returns a mapping between functions of two signatures. This property allows us to address the case when an unknown file is a bundle file containing multiple libraries. Since with this information we will not attempt to match a function that was previously matched.

The last contribution is an evaluation of the algorithm and an evaluation of a large corpus of hybrid Android applications against all known to be vulnerable libraries from the npm repository. This evaluation allows us to establish the method of matching signature that works the best compared to other methods. Additionally, we discovered some of the

limitations of our approach as well as difficulties with the gathering of data. Such as some difficulties in the case of matching minified source code versus non-minified source code, and the limitations coming from the fact we are using only ten candidates for each JS file.

6.2 Future Work

The first item that could be done in the future is related to the improvement of the signature that is created from the JS file. Improving this will allow us to detect more libraries used among Android applications. We should ensure that function tokens that do not bring any useful information about the function are not included into the function signature. For example, some variable names are recorded in the signature, and some function calls are recording the name of the function that is being called. It is possible that these data points prevent some matches when we are trying to detect the minified version of the unknown file against the non minified version of the library. Because of this, it might be possible to remove some of those tokens from signatures. However, a thorough evaluation is required to make sure that the removal of signatures does not cause the matches to be missed.

Another item is related to the improvement of partial matches throughout various parts of the system. Across most matching methods there are two places where the similarity metrics (Jaccard Like Index and Lib Portion Index, both described in Section 4.2.4) are being applied. One of the places is when the function signature is compared to all other functions to create the best candidate for a given function. The second place is when we are creating the final similarity value between two files, after all functions got best possible matches. Two of the methods that we define — `fn-st-toks-v5` and `fn-st-toks-v6` — only use exact matches between functions during the first comparison between functions. However, for the last comparison producing the similarity between two files, these two matching methods report the similarity value as it is — whether it is exact or partial match value. We have attempted to use partial matches in methods `fn-st-toks-v1`, `fn-st-toks-v2`, `fn-st-toks-v3`, and `fn-st-toks-v4`. In the method `fn-st-toks-v4` we also attempted to include similarity values between functions into the calculation of the final similarity value between files. However, none of those methods performed as well as `fn-st-toks-v6` as seen from Tables 5.2 and 5.3.

Perhaps, we can attempt to improve partial matches between functions, by giving priority to exact matches and then allowing partial matches for the remaining functions when there are no other options. This two stage technique is similar to Algorithm 2, where we are trying to find as many exact matches before matching remaining libraries as is. The difference with Algorithm 2 is that Algorithm 2 is giving higher priority to the first

part with exact matches and only records partial matches in the second part when there are no other options left available. However, with the information about partial function matches in place, we might be able to reflect it in the final similarity metric between files along the lines of the `fn-names-st-toks` method. Additionally, with improvements to the handling of partial function matches as part of the matching methods, we should be able to see improved results created in the second part of Algorithm 2.

Additionally, we can also perform an analysis similar to the work done in the recently released paper by Ponta et al. [36] where they describe an approach combining static and dynamic analysis to determine the reachability of various known vulnerabilities. For their scenario, they have the information about vulnerable sections of the library, which we do not have. We would have to collect locations of vulnerabilities to compile the source of truth. Additionally, they derive methods to determine the best possible version of a library to upgrade to, which should translate quite easily to JS, making this idea a useful tool for developers.

Lastly, as one of the ways to address the problem we can employ the approach described by Mirhosseini et al. [24]. The researchers analysed the impact of notification services that keep track of outdated dependencies and either show the developer the badge with the status of freshness of their dependencies, or submit a pull request on behalf of the service updating those outdated dependencies. The paper analysed 7470 GitHub projects that used such services to detect any change in the behavior compared to projects not using those services. They find that, on average, projects using automated pull requests updated 1.6x more frequently, and projects using notification badges updated 1.4x more frequently, compared to projects that did not use any of these notification services. They also find that the notification services add to developers' notification fatigue and they suggest ways about improving notifications in such a way that they can actually help developers.

6.3 Concluding Remarks

Mobile and Web platforms are ever growing platforms. More and more applications and web sites are created, as well as various developer tools to aid the end-user developer. However, with the popularity of those platforms, the research aimed to find problems with these platforms is lagging behind. The Android platform has seen attention from both the security and the software engineering community. There are both papers discussing issues with the platform as well as ways to solve some of these problems. That is boosted by the fact that the Android platform uses a strongly typed language allowing for simpler static analyses. On the other hand, the Web platform is using a very dynamic language,

which does not aid researchers in simple analyses. Because of this, the majority of papers published so far among the security community is about finding vulnerabilities in the Web platform and some suggestions about reducing the impact of those vulnerabilities. On the other hand, there is not a lot of research done in the software engineering community with dynamic languages in general.

In this thesis we attempt to evaluate the quality of the Android applications that use Web technologies. We attempt to detect known to be vulnerable libraries that are used among hybrid applications on the Android platform. Such a tool would aid end-users as well as owners of application repositories in determining whether some application is safe to be used or if it should be taken care of. It may also help researchers who are pursuing similar goals, analysing dynamic languages. This tool is a stepping stone towards the analysis of dynamic languages, such as JavaScript in this case. With the items outlined in the future work, we hope that this tool will be useful to a wide range of people, when working with JavaScript based projects.

References

Papers and Conferences

[12] Julius Davies, Daniel M. German, Michael W. Godfrey, and Abram Hindle. "Software Bertillonage: Finding the Provenance of an Entity". In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. MSR '11. Waikiki, Honolulu, HI, USA, May 22, 2011. ISBN: 978-1-4503-0574-7. DOI: 10.1145/1985441.1985468. URL: http://dl.acm.org/citation.cfm?id=1985468 (visited on 10/12/2017) (cit. on p. 16).

[13] Alexandre Decan, Tom Mens, and Eleni Constantinou. "On the Evolution of Technical Lag in the Npm Package Dependency Network". In: *Proceedings of the 34th International Conference on Software Maintenance and Evolution*. ICSME '18. Madrid, Spain, Sept. 23, 2018. ISBN: 978-1-5386-7870-1. DOI: 10.1109/ICSME.2018.00050. URL: https://ieeexplore.ieee.org/document/8530047 (visited on 11/25/2018) (cit. on p. 14).

[20] Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. "Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web". In: *Proceedings of the 24th Annual Network and Distributed System Security Symposium*. NDSS '17. San Diego, CA, USA, Feb. 27, 2017. ISBN: 978-1-891562-46-4. DOI: 10.14722/ndss.2017.23414. URL: https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/thou-shalt-not-depend-me-analysing-use-outdated-javascript-libraries-web/ (visited on 03/07/2018) (cit. on pp. 2, 6, 13, 15, 23).

[21] Sebastian Lekies and Martin Johns. "Lightweight Integrity Protection for Web Storage-Driven Content Caching". In: *Proceedings of the 6th Workshop on Web 2.0 Security & Privacy*. W2SP '12. San Francisco, CA, USA, May 24, 2012. URL: https:

//www.ieee-security.org/TC/W2SP/2012/papers/w2sp12-final8.pdf (visited on 11/25/2018) (cit. on p. 17).

[22] Sebastian Lekies, Ben Stock, and Martin Johns. "25 Million Flows Later: Large-Scale Detection of DOM-Based XSS". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. CCS '13. Berlin, Germany, Nov. 4, 2013. ISBN: 978-1-4503-2477-9. DOI: 10.1145/2508859.2516703. URL: http://doi.acm.org/10.1145/2508859.2516703 (visited on 10/28/2018) (cit. on p. 18).

[23] Zhou Li, Sumayah Alrwais, XiaoFeng Wang, and Eihal Alowaisheq. "Hunting the Red Fox Online: Understanding and Detection of Mass Redirect-Script Injections". In: *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. SP '14. San Jose, CA, USA, May 18, 2014. ISBN: 978-1-4799-4686-0. DOI: 10.1109/SP.2014.8. URL: https://doi.org/10.1109/SP.2014.8 (visited on 10/29/2018) (cit. on p. 18).

[24] Samim Mirhosseini and Chris Parnin. "Can Automated Pull Requests Encourage Software Developers to Upgrade Out-of-Date Dependencies?" In: *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. ASE '17. Urbana-Champaign, IL, USA, Oct. 30, 2017. ISBN: 978-1-5386-2684-9. URL: http://dl.acm.org/citation.cfm?id=3155562.3155577 (visited on 10/25/2018) (cit. on p. 67).

[25] Patrick Mutchler, Adam Doupé, John Mitchell, Chris Kruegel, and Giovanni Vigna. "A Large-Scale Study of Mobile Web App Security". In: *Proceedings of the Workshop on Mobile Security Technologies*. MoST '15. San Jose, CA, USA, May 21, 2015. URL: https://www.ieee-security.org/TC/SPW2015/MoST/papers/s2p3.pdf (visited on 09/26/2018) (cit. on p. 15).

[34] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. "Vulnerable Open Source Dependencies: Counting Those That Matter". In: *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '18. Oulu, Finland, Oct. 11, 2018. ISBN: 978-1-4503-5823-1. DOI: 10.1145/3239235.3268920. URL: http://doi.acm.org/10.1145/3239235.3268920 (visited on 11/10/2018) (cit. on p. 13).

[36] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. "Beyond Metadata: Code-Centric and Usage-Based Analysis of Known Vulnerabilities in Open-Source Software". In: *Proceedings of the 34th International Conference on Software Maintenance and Evolution*. ICSME '18. Madrid, Spain, Sept. 23, 2018. ISBN: 978-1-5386-7870-1. DOI: 10.1109/ICSME.2018.00054. URL: https://ieeexplore.ieee.org/document/8530051 (visited on 11/25/2018) (cit. on pp. 14, 67).

[38] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. "The Eval That Men Do: A Large-Scale Study of the Use of Eval in Javascript Applications". In: *Proceedings of the 25th European Conference on Object-Oriented Programming*. ECOOP '11. Lancaster, UK, July 25, 2011. ISBN: 978-3-642-22654-0. URL: http://dl.acm.org/citation.cfm?id=2032497.2032503 (visited on 10/29/2018) (cit. on p. 18).

[39] Y. Sabi, Y. Higo, and S. Kusumoto. "Rearranging the Order of Program Statements for Code Clone Detection". In: *Proceedings of the 11th International Workshop on Software Clones*. IWSC '17. Klagenfurt, Austria, Feb. 21, 2017. ISBN: 978-1-5090-6595-0. DOI: 10.1109/IWSC.2017.7880503. URL: https://ieeexplore.ieee.org/document/7880503 (visited on 11/25/2018) (cit. on p. 16).

[40] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. "SourcererCC: Scaling Code Clone Detection to Big-Code". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas, May 14, 2016. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884877. URL: http://doi.acm.org/10.1145/2884781.2884877 (visited on 10/12/2017) (cit. on p. 15).

[42] Sooel Son and Vitaly Shmatikov. "The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites". In: *Proceedings of the 20th Annual Network and Distributed System Security Symposium*. NDSS '13. San Diego, CA, USA, Apr. 23, 2013. URL: https://www.ndss-symposium.org/ndss2013/ndss-2013-programme/postman-always-rings-twice-attacking-and-defending-postmessage-html5-websites/ (visited on 11/25/2018) (cit. on p. 17).

Other Sources

[1] *apktool project page*. URL: https://ibotpeaches.github.io/Apktool (visited on 08/22/2018) (cit. on pp. 12, 23).

[2] Lakshmanan Arumugam. *How we mine Google Play Data*. URL: https://laksh47.github.io/crawler-demo/ (visited on 12/12/2018) (cit. on p. 23).

[3] Abel Avram. *Creating Mobile Native Apps in JavaScript with NativeScript*. Mar. 6, 2015. URL: https://www.infoq.com/news/2015/03/nativescript (visited on 09/18/2018) (cit. on p. 5).

[4] *babel minify project page*. URL: https://github.com/babel/minify (visited on 08/22/2018) (cit. on p. 7).

[5] *babylon JS parser project page*. URL: https://github.com/babel/babylon (visited on 11/27/2018) (cit. on p. 29).

[6] Adam Baldwin. *Resources Downloaded over Insecure Protocol, Affecting chromedriver126, ALL versions*. Jan. 4, 2017. URL: https://snyk.io/vuln/npm:chromedriver126:20170101 (visited on 11/10/2018) (cit. on p. 61).

[7] *bower official site*. URL: https://bower.io (visited on 08/22/2018) (cit. on p. 33).

[8] *componentjs project page*. URL: https://github.com/componentjs/component (visited on 08/22/2018) (cit. on p. 33).

[9] *Cordova official site*. URL: https://cordova.apache.org (visited on 08/22/2018) (cit. on p. 1).

[10] *Crawler demo project page*. URL: https://github.com/Laksh47/crawler-demo (visited on 12/12/2018) (cit. on p. 23).

[11] *Crawler project page*. URL: https://github.com/uw-swag/swag-crawler (visited on 12/12/2018) (cit. on p. 23).

[14] *electron project page*. URL: https://electronjs.org (visited on 09/18/2018) (cit. on p. 5).

[15] Liang Gong. *Directory Traversal, Affecting cyber-js, ALL versions*. June 7, 2017. URL: https://snyk.io/vuln/npm:cyber-js:20170418 (visited on 10/18/2018) (cit. on p. 57).

[16] Liang Gong. *Directory Traversal, Affecting zhangranbigman, ALL versions*. Feb. 26, 2018. URL: https://snyk.io/vuln/npm:zhangranbigman:20180226 (visited on 10/18/2018) (cit. on p. 60).

[17] *google closure compiler project page*. URL: https://github.com/google/closure-compiler-js (visited on 08/22/2018) (cit. on p. 7).

[18] *Ionic framework site*. URL: https://ionicframework.com (visited on 11/22/2018) (cit. on p. 11).

[19] Paul Jaccard. "The Distribution of the Flora in the Alpine Zone". In: *New Phytologist* 11.2 (Feb. 1912), pp. 37–50. ISSN: 0028-646X. DOI: 10.1111/j.1469-8137.1912.tb05611.x. URL: https://nph.onlinelibrary.wiley.com/doi/abs/10.1111/j.1469-8137.1912.tb05611.x (visited on 11/22/2018) (cit. on p. 36).

[26] *nodesecurity project page*. URL: https://nodesecurity.io (visited on 08/22/2018) (cit. on p. 21).

[27] *npm official site*. URL: https://npmjs.com (visited on 08/22/2018) (cit. on pp. 1, 21).

[28] *npm package chromedriver*. URL: https://www.npmjs.com/package/chromedriver (visited on 10/18/2018) (cit. on p. 61).

[29] *npm package chromedriver126*. URL: https://www.npmjs.com/package/chromedriver126 (visited on 10/18/2018) (cit. on p. 61).

[30] *npm package cyber-js*. URL: https://www.npmjs.com/package/cyber-js (visited on 10/18/2018) (cit. on p. 57).

[31] *npm package zhangranbigman*. URL: https://www.npmjs.com/package/zhangranbigman (visited on 10/18/2018) (cit. on p. 60).

[32] *npm rank — top 1000 most dependent-upon packages*. URL: https://gist.github.com/anvaka/8e8fa57c7ee1350e3491 (visited on 01/28/2018) (cit. on p. 22).

[33] Tom Occhino. *React Native: Bringing modern web techniques to mobile*. Mar. 26, 2015. URL: https://code.fb.com/android/react-native-bringing-modern-web-techniques-to-mobile (visited on 09/18/2018) (cit. on p. 5).

[35] *PhoneGap official site*. URL: https://phonegap.com (visited on 11/22/2018) (cit. on p. 11).

[37] *React-Native official site*. URL: https://facebook.github.io/react-native (visited on 08/22/2018) (cit. on p. 1).

[41] *snyk project page*. URL: https://snyk.io (visited on 08/22/2018) (cit. on p. 21).

[43] Darryl K. Taft. *PhoneGap Simplifies iPhone, Android, BlackBerry Development*. Ed. by eweek.com. Mar. 13, 2009. URL: http://www.eweek.com/development/phonegap-simplifies-iphone-android-blackberry-development (visited on 09/18/2018) (cit. on p. 5).

[44] *The State of the Octoverse 2017*. URL: https://octoverse.github.com (visited on 08/22/2018) (cit. on pp. 1, 11).

[45] *UglifyJS project page*. URL: https://github.com/mishoo/UglifyJS2 (visited on 08/22/2018) (cit. on p. 7).

[46] Matt Weinberger. *One programmer almost broke the internet by deleting 11 lines of code*. Mar. 23, 2016. URL: https://www.businessinsider.com/npm-left-pad-controversy-explained-2016-3 (visited on 11/07/2018) (cit. on p. 64).

[47] Chris Williams. *How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript*. Mar. 23, 2016. URL: https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos (visited on 11/07/2018) (cit. on p. 64).