Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

Nadine Laube
MASTER THESIS

URL2SBOM

Identification of Client-Side JavaScript Libraries

Submitted on 15.04.2024

Supervisor: Prof. Dr. Dirk Riehle, M.B.A.,
M.Sc. Martin Wagner
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander University Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 15.04.2024

License

This work is licensed under the Creative Commons Attribution 4.0 International license (CC BY 4.0), see https://creativecommons.org/licenses/by/4.0/

Erlangen, 15.03.2024

Abstract

This thesis concerns itself with the identification of third-party JavaScript libraries used on websites. To identify the libraries and their version from client-side scripts, which are likely to be slightly modified, minified, obfuscated or bundled together, a corpus of popular client-side JavaScript libraries is constructed. The entry point for each library is determined by the *jsDelivr* API and the corresponding file is downloaded in minified form and if available also in non-minified form. These files are then used as input for *Siamese (Ragkhitwetsagul & Krinke, 2019)*, constructing multiple representations and indexing them for later queries in *ElasticSearch*. *Siamese* performs static analysis comparing multiple representations of source code to identify potential code clone pairs. A small proto-benchmark consisting of four popular JavaScript libraries in multiple transformed variations is created to evaluate *Siamese*'s suitability for the task of code clone search on JavaScript libraries to match them to the corpus for their identification. The testing on said benchmark revealed that *Siamese* is not able to process all the supplied input data correctly, often failing quietly. The tool is also inherently susceptible to obfuscated code.

Contents

1 Introduction ... 8
1.1 Fundamentals ... 9
1.1.1 Basic Information about JavaScript ... 9
1.1.2 Minification, Obfuscation, Transpilation and Bundling ... 11
1.1.3 Code Similarity and Code Clones ... 16
1.2 Motivation ... 17
1.3 Objective ... 20
1.4 Thesis Structure ... 20
2 Literature Review ... 21
3 Requirements ... 25
3.1 Functional Requirements ... 25
3.2 Non-Functional Requirements ... 26
4 Architecture ... 27
4.1 Research Conclusions ... 27
4.2 Siamese ... 29
4.3 Corpus Builder ... 31
5 Design and Implementation ... 34
5.1 Design ... 34
5.1.1 Assumptions ... 34
5.1.2 Resulting Design Choices ... 35
5.2 Implementation of the Corpus Builder ... 35
5.2.1 Download ... 36
5.2.2 Transform ... 36
5.2.3 Execute ... 37
5.3 Leveraging Siamese for Code Clone Identification ... 38
5.3.1 Configuration ... 38
5.3.2 Other Problems ... 39
5.4 Benchmark ... 39
5.4.1 General Considerations for Creating a Benchmark ... 39
5.4.2 Creating a JavaScript Benchmark for Code Clone Identification ... 40
6 Evaluation ... 43
6.1 Results on the Benchmark ... 43
6.2 Evaluation of Functional Requirements ... 46
6.3 Evaluation of Non-Functional Requirements ... 48
7 Conclusions ... 49
7.1 Conclusions ... 49
7.2 Threats to Validity ... 50
7.3 Future Work ... 51
References ... **Fehler! Textmarke nicht definiert.**

Index of Figures

Figure 1: Overview of the SpiderMonkey browser engine. From Guerra Lourenço, 2023 10

Figure 2: A short JavaScript program containing multiple comments 12

Figure 3: Minified (by terser-c -m v5.16.5) version of Figure 2, with only a single remaining comment – manually re-added whitespace for human readability... 13

Figure 4: Obfuscated (by javascript-obfuscator v4.0.0, enabled settings: transform object keys, numbers to expressions, control flow flattening, hexadecimal identifier name generator, seed=0) version of Figure 2.. 14

Figure 5: AST for original code, as shown in Figure 2, parsed by acorn................................ 15

Figure 6: AST for minified code, as shown in Figure 3, parsed by acorn............................... 16

Figure 7: Combining the corpus builder and Siamese for clone search.................................. 27

Figure 8: Using Moss to compare two obfuscated variations of the same code snippet.......... 28

Figure 9: Siamese architecture. From Ragkhitwetsagul & Krinke, 2019 29

Figure 10: The four representations generated by Siamese 30

Figure 11: UML of the download package... 31

Figure 12: UML of the transform package... 31

Figure 13: UML of the execute package 32

Figure 14: Simplified Sequence Diagram of the Corpus Builder Application......................... 33

Index of Tables

Table 1: JavaScript libraries and transformations performed for the benchmark 42

Table 2: Siamese’s top 8 query result matrix 44

Table 3: Output for the query of obfuscated jquery@1.5.1 depending on Siamese and ElasticSearch settings.. 45

Table 4: Siamese’s top 8 query results for jquery 46

Table 5: Siamese’s top 8 query results for whatwg-fetch.. 46

Abbreviations

ANTLR	ANother Tool for Language Recognition
AST	Abstract Syntax Tree
BCE	Byte-Code Emitter
CDN	Content Delivery Network
CLI	Command-Line Interface
CSS	Cascading Style Sheet
CSV	Comma-Separated Values
DOM	Document Object Model
GCF	General Clone Format
GenAI	Generative Artificial Intelligence
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
JIT	Just-In-Time
LCS	Longest Common Subsequence
LSH	Locality Sensitive Hashing
NPM	Node Package Manager
PDG	Programm Dependency Graph
RCF	Rich Clone Format
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
SBOM	Software Bill of Materials
SVG	Scalable Vector Graphics
TF-IDF	Term Frequency - Inverse Document Frequency

1 Introduction

Copying all or parts of a program is as natural to a programmer as breathing, and as productive. It ought to be as free. – Richard M. Stallmann

It is common practice for software developers to use free open-source components like libraries, or even to copy and paste pieces of source code instead of writing the code themselves to avoid "reinventing the wheel" and to boost productivity. But with the use of open-source software also comes the commitment to honour the license agreements that often come with obligations, e.g., proper attribution or having to disclose source code, attached to the distribution of software.

Rosen (2004, p. 42) provides a straightforward definition of software distribution:

> *[Software distribution] means selling or giving copies of software away to others. It also may include such arrangements as incorporating software into consumer or industrial products and selling those products to others. For some software, it may also include making the software available across a network for execution by others.*

A website is therefore a form of distribution of software, as the JavaScript code hosted on a server is made available via network and then executed on the client's computer by their browser.

The practice of copy-paste-edit results in so-called code clones. While this can be done out of expediency to meet deadlines, it can introduce technical debt as removing code clones later leads to invasive refactoring of the system posing a risk to its stability. Godfrey and Kapser (2021) argue that while the risks of code cloning include code bloat, increased bugginess, creeping system fragility as well as inconsistent maintenance, not all code clones are harmful but sometimes even beneficial to the long-term health of a software system. Based on the developers' intentions they define three meta-groups:

- **Forking**: developing similar solutions within a new context, including having variations for different hardware, having co-existing versions for platform variations, or branching, to work on experimental functionalities without endangering the stability of the core system
- **Templating**: mostly caused by the underlying programming language and its lack of abstraction mechanisms; including parameterized code, boilerplating, and programming idioms but also repeated API calls following a protocol
- **Customization**: opportunistic reuse of existing code that solves a similar problem, including clone-and-own practices or bug workarounds, when bugs are difficult to fix at the source due to ownership issues or unacceptable exposure risk

While code clones created with the intention of forking or customization will naturally show some divergence over time, code clones with the intent of templating often require to be closely maintained together. The mentioned inconsistent maintenance could look like the following: Patches that became available in the original open-source component need to be manually applied in the local copy, changes to duplicate code need to be made to all occurrences in the codebase, and previously working functionalities might have been deprecated. If not well documented and maintained, this can cause serious problems as problematic code, malicious or vulnerable, might also flow into downstream projects. (Kim, Woo, Lee, & Oh, 2017; Monden, Nakae, Kamiya, Sato, & Matsumoto, 2002)

Code clones that are not well-documented and properly attributed could also cause copyright disputes (Wu, Manabe, Kanda, German, & Inoue, 2015). A recent example of disputes caused by non-compliance with open source licenses includes the lengthy French legal battle between

the company Entr'Ouvert and the telecommunications provider Orange over Orange not honouring the GNU GPL v2 license agreements when using their software *LASSO*. After almost 13 years of litigation, the Court of Appeal of Paris ordered Orange to pay 650.000€ for their violations of the GNU GPL license in February 2024. (Noisette, 2024)

1.1 Fundamentals

In this chapter, the necessary knowledge to follow this thesis will be provided. First, a brief overview of JavaScript and its peculiarities will be provided, also the difficulties of working with client-side code will be highlighted and the basic concepts of code similarity and code clones will be explained.

1.1.1 Basic Information about JavaScript

ECMAScript, the core language specification of JavaScript, is standardised by the ECMA TC39 Committee. After the publication of the third edition, ES3 in 2002, ECMAScript started to become massively adopted as the programming language of the web being supported by essentially all web browsers (ECMA TC39 Committee).

But not all JavaScript code is created equal, it makes a difference for which runtime it is written, as there exist multiple inconsistencies on features additional to the ECMAScript specifications: Code written for Node.js (OpenJS Foundation) or Deno (Dahl, justjavac, & et al, 2018) will look very different from code written for browsers. While runtimes like Node.js natively support CommonJS modules (using the *require* keyword), browsers only support ECMAScript modules natively (using the *import* keyword) (Haverbeke, 2019). By employing build tools such as *webpack* (Koppers, Ewald, Larkin, & Kluskens, 2016) scripts containing CommonJS modules can be transformed to browser compatibility.

One of the oldest and most popular package managers in the JavaScript ecosystem is *Node Package Manager* (NPM) (npm, Inc., 2009), which manages dependencies in the *package.json* file. This file is used during the development and build process, but it is not deployed with the website. Similar documents are often produced by build tools such as *webpack*.

In this thesis, the focus will be on client-side JavaScript code executed by browsers.

According to the specification of HTML (WHATWG, 2024), there are multiple common ways to add JavaScript code to websites:

JavaScript code can be added to websites via the script element, either as an inline script, with the code embedded as text in the script element, or as an external script, specifying the location of the source code. External scripts can either be hosted on the same (sub)domain as the website or a completely different domain, e.g. Content Delivery Networks (CDN) or other providers of third-party components. The resolution of import statements can also be specified via an import map, defined once per document as an inline script element with its type attribute set to "importmap". Event handlers, e.g., *onclick*, *onmousemove* etc, can also contain JavaScript code that will be invoked when the corresponding event is observed. Interestingly, SVG elements can also contain both script tags and event handlers, and can therefore be used to invoke the execution of JavaScript code. Since JavaScript can evaluate code from strings using functions such as *eval()*, regular variables or string literals can also contain code.

The nodes and elements of a web page form a tree, called the Document Object Model (DOM), which can be accessed and manipulated by the scripts included in the page.

The code executed for a given website can be browser-specific with different features activated to circumvent cross-browser incompatibilities or performance issues (Richards, Gal, Eich, & Vitek, 2011).

Lauinger et al. (2017) found that inline scripts – including script elements, event handlers and code evaluated from strings – were the most common script type. They also investigated external scripts, discovering that 91.7% of inspected sites included at least one script originating from domains that were neither the sites' own domain nor subdomain.

Mitropoulos, Louridas, Salis, and Spinellis (2019) studied the evolution of client-side JavaScript applications over a period of nine months, observing a high development pace as websites seem to change constantly. Only 10% of inspected sites remained without changes through the whole study period; The mean time scripts remained unchanged was reported at 7 days for inline scripts and only 5 days for external JavaScript files.

JavaScript is a dynamic, mostly untyped and asynchronous language, making it very difficult to construct call graphs. Static analysis of JavaScript source code struggles to capture dynamic calls from non-trivial use of calls such as *eval()*, *bind()*, or *apply()*, while dynamic analysis' completeness is highly dependent on the quality of test cases, as only executed code can be captured by this approach. While multiple tools exist for creating static call graphs of JavaScript, the levels of effectiveness and coverage of newer ECMAScript standards vary. (Antal, Hegedűs, Herczeg, Lóki, & Ferenc, 2023)

Browsers rely on Just-In-Time (JIT) compilers to execute JavaScript code. In Figure 1 the process of executing JavaScript is outlined for the JavaScript engine *SpiderMonkey* of the Mozilla Firefox browser. The core structure – shared by other JavaScript engines – usually includes a compiler infrastructure with one or more JIT compilers, a virtual machine operating JS values and includes a bytecode interpreter, as well as a runtime that provides a set of native objects and functions (Guerra Lourenço, 2023).

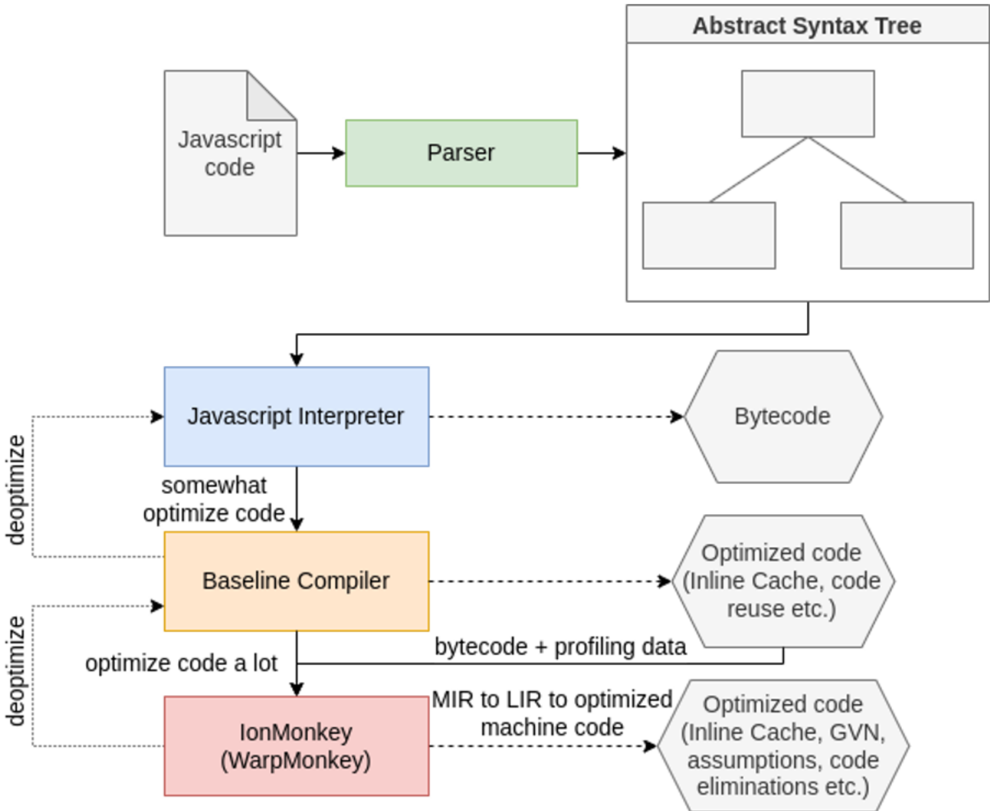

Figure 1: Overview of the SpiderMonkey browser engine. From Guerra Lourenço, 2023

According to the *SpiderMonkey* documentation (Mozilla Foundation), the first step, as depicted in Figure 1, is parsing the JavaScript code to an Abstract Syntax Tree (AST):

An AST is a hierarchical representation of the structure and relationships of tokens – literals, variables, operators etc. – in the code and can be obtained by lexing and parsing a source code, typically performed by a compiler or interpreter (Princeton University Department of Computer Science, 2022). Based on *SpiderMonkey* (Mozilla Foundation), a community JavaScript AST representation standard called ESTree (ESLint, Acorn, Babel, & Mozilla Foundation) evolved.

The parser is then running a Byte-Code Emitter (BCE). By default, the parser runs in lazy mode, avoiding generating the full AST or byte code for the source code that is being parsed, saving CPU time and memory. Next, the baseline interpreter – a hybrid interpreter/JIT – interprets one opcode at a time, speeds up executing the same opcode next time by employing inline caching. Then the baseline compiler uses the same inline cache mechanism and translates the byte code to machine code, adding local optimizations. Lastly, *WarpMonkey* – formerly *IonMonkey* – offers the highest level of optimization for frequently run scripts, transforming the byte code and inline cache to a mid-level intermediate representation, applying optimization strategies, e.g. pruning unused branches, and then transforming it into a low-level intermediate representation for register allocation and code generation. (Mozilla Foundation)

1.1.2 Minification, Obfuscation, Transpilation and Bundling

To improve the performance of websites, tricks like reducing file size to minimize latency when fetching data over a network or combining multiple files into one for fewer requests over the network are employed: To reduce the size of JavaScript code minifier tools are applied, removing comments and whitespace, renaming bindings and even replacing code with shorter equivalents. The concatenation of multiple files can be done manually or automatically by bundlers. (Haverbeke, 2019)

According to Zammetti (2022), *webpack* (Koppers et al., 2016) has emerged as the de facto standard bundler. It takes a given entry and project directory, constructs a dependency graph, finds out what needs to be included and what can be dropped, concatenates the relevant files and resolves conflicts, e.g. naming collisions. The resulting file, called a bundle, can also contain more than just source code: It can also include other resources like CSS or images encoded as data URLs. *Webpack* has plenty more features, including transpilers for *.jsx or *.tsx files, optimizing bundles by tree shaking, minification and compression, and abundantly more can be added via plugins. The technique of tree shaking, i.e. the removal of unused code, was further optimized by Vázquez, Bergel, Vidal, Díaz Pace, and Marcos (2019), removing unused functions from JavaScript libraries and therefore reducing the bundle size.

Code obfuscation is a technique to hide the underlying logic of the application, making reverse engineering or attacking an application more difficult. It does not change the functionality of the code but changes the representation to make it less readable or recognizable. Obfuscation techniques can transform the layout of the source code, the data, structures, i.e. methods or classes, or even the control flow. (Collberg, Thomborson, & Low, 1997; Huang et al., 2023)

Skolka, Staicu, and Pradel (2019) conducted an empirical study on minified and obfuscated code on the web and found that code transformations are very common, affecting 38% of all scripts. The majority of affected scripts are minified and only 1% of all scripts are transformed by advanced obfuscation techniques.

Nicolini, Hora, and Figueiredo (2024) investigated newly proposed JavaScript features and their usage in software projects. They found that 73 of 1000 projects used at least one proposed feature that was not yet part of the ECMAScript specification. They also found that new features have an average of 86% compatibility across the most commonly used browsers, creating a lack of compatibility for 14% of browsers that require additional tools to bridge the gap: A tool that takes source code and generates syntactically equivalent source code in another target language is called a transpiler. With the ECMAScript specification evolving faster than some runtimes, JavaScript developers can use transpilers to utilize novel features while remaining compatible with outdated runtimes, e.g. older browser versions, by converting their code to an old JavaScript syntax. The most popular transpiler for JavaScript is *Babel* (McKenzie & et al, 2018), which transforms code to a version compatible with older browsers, typically resulting in code equivalent to ES5, and adds features missing in the target environment via polyfills.

Multiple variations of JavaScript with additional features such as strict data typing or syntactic sugar exist, e.g. TypeScript or CoffeeScript, that can easily be transpiled to regular JavaScript code to make the development of web applications easier.

To illustrate what these modifications would look like on JavaScript code, Figure 2 contains some regular JavaScript source code with an implementation of a function calculating Fibonacci numbers, some unused code and four comments, three of them marked “@preserve”.

```
// possible license comment 1
function fib(number) {
    // @preserve possible license comment 2
    function mult(a, b) {
        return a * b;
    }
    function sum(a, b) {
        return a + b;
    }

    if (!Number.isInteger(number) || !(number >= 0)) {
        throw `fib: number is ${number}. ` +
              "positive integer expected.";
    }

    switch (number) {
    case 0:
        // @preserve possible license comment 3
        return number;
        break;
    case 1:
        // @preserve possible license comment 4
        return number;
        break;
    }

    return sum(fib(number-1), fib(number-2));
}
```

Figure 2: A short JavaScript program containing multiple comments

In Figure 3 the previously introduced source code has been minified using the tool *terser (Santos & Vicente, 2018)* version 5.16.5 with the options compress and mangle activated. The code is much shorter and would contain no linebreaks in its original form, they were manually re-added for improved readability and easier comparison. The function *mult*, being unused code, has been removed completely, case 0 is now implemented by fallthrough. The function *sum* is now inlined, as it is only used in the return statement. The name of the variable *number* was shortened to ‘e’ and even the concatenation of a string template and a string has been evaluated to only one concatenated string. Also, the logical expression of the if-statement was reformulated by applying De Morgan’s law to shorten the expression by 1 character.

```
function fib(e){
    if(!(Number.isInteger(e)&&e>=0))
        throw`fib: number is ${e}. positive integer expected.`;

    switch(e){
        case 0:
        case 1:
            // @preserve possible license comment 4
            return e
    }

    return i=fib(e-1),r=fib(e-2),i+r;var i,r
}
```

Figure 3: Minified (by terser-c -m v5.16.5) version of Figure 2, with only a single remaining comment – manually re-added whitespace for human readability

Terser’s documentation (Santos & Vicente, 2023) states that comments marked “@preserve” or “@license” are usually not removed, unlike regular comments. But in this case, only one comment remains, as the other comments, e.g. “possible license comment 2”, were attached to unused code or code that was optimized away and therefore dropped.

To demonstrate the effects of obfuscation, the source code has been obfuscated using *javascript-obfuscator* v4.0.0 (Kachalov, 2016) with the options for transforming object keys, and numbers to expressions and control flow flattening enabled. The names of functions have purposefully not been changed to make the comparison between the obfuscated source code in Figure 4 and the original source code in Figure 2 easier. Unlike the minification performed in Figure 3, the code obtained by obfuscation is longer than the original, it contains seven additional lines of code. While the definition of the function *sum* remains identical, everything else has changed: A new variable *wvX01n* has been introduced. With the structure of a dictionary, five functions are defined as values to random-looking key strings. One of the functions is a simple multiplication of both arguments, now called by the *mult* function instead of directly implementing it. Similarly, new functions have been added to wrap the function calls in the return statement. Numbers, such as the numbers in the return statement or the case clauses of the switch/case, have been obfuscated by transforming them into a calculation of hexadecimal numbers. Additionally, the whitespace in strings has also been changed to a hexadecimal encoding, replacing the space character with ‘\x20’. The – albeit light – obfuscation has resulted in hardly legible source code.

```
function fib(number) {
    var wvXOln = {
        'WbfrN': function (x, y) {
            return x * y;
        },
        'GojsA': function (x, y) {
            return x >= y;
        },
        'WmSaF': function (callee, param1, param2) {
            return callee(param1, param2);
        },
        'Gbwui': function (callee, param1) {
            return callee(param1);
        },
        'pQQKm': function (x, y) {
            return x - y;
        }
    };
    function mult(a, b) {
        return wvXOln['WbfrN'](a, b);
    }
    function sum(a, b) {
        return a + b;
    }
    if (!Number['isInteger'](number) || !wvXOln['GojsA'](number, -0xe7a +
0x8cc + -0x2 * -0x2d7)) {
        throw 'fib:\x20number\x20is\x20' + number + '.\x20' +
'positive\x20integer\x20expected.';
    }
    switch (number) {
    case -0x123 * -0xa + -0x9e * 0x28 + 0xd52:
        return number;
        break;
    case -0x3b * -0xf + 0x2159 * -0x1 + -0x3 * -0x9f7:
        return number;
        break;
    }
    return wvXOln['WmSaF'](sum, wvXOln['Gbwui'](fib, wvXOln['pQQKm'](number, -
0x843 + -0x29 * 0xab + 0x23a7)), wvXOln['Gbwui'](fib, number - (0x26b0 + -0x1
* 0x180b + -0xea3 * 0x1)));
}
```

Figure 4: Obfuscated (by javascript-obfuscator v4.0.0, enabled settings: transform object keys, numbers to expressions, control flow flattening, hexadecimal identifier name generator, seed=0) version of Figure 2

The transformation of minifying code also affects the AST representation of the code, Figure 5 shows the AST generated for the regular code in Figure 2 while Figure 6 shows the AST after the minification.

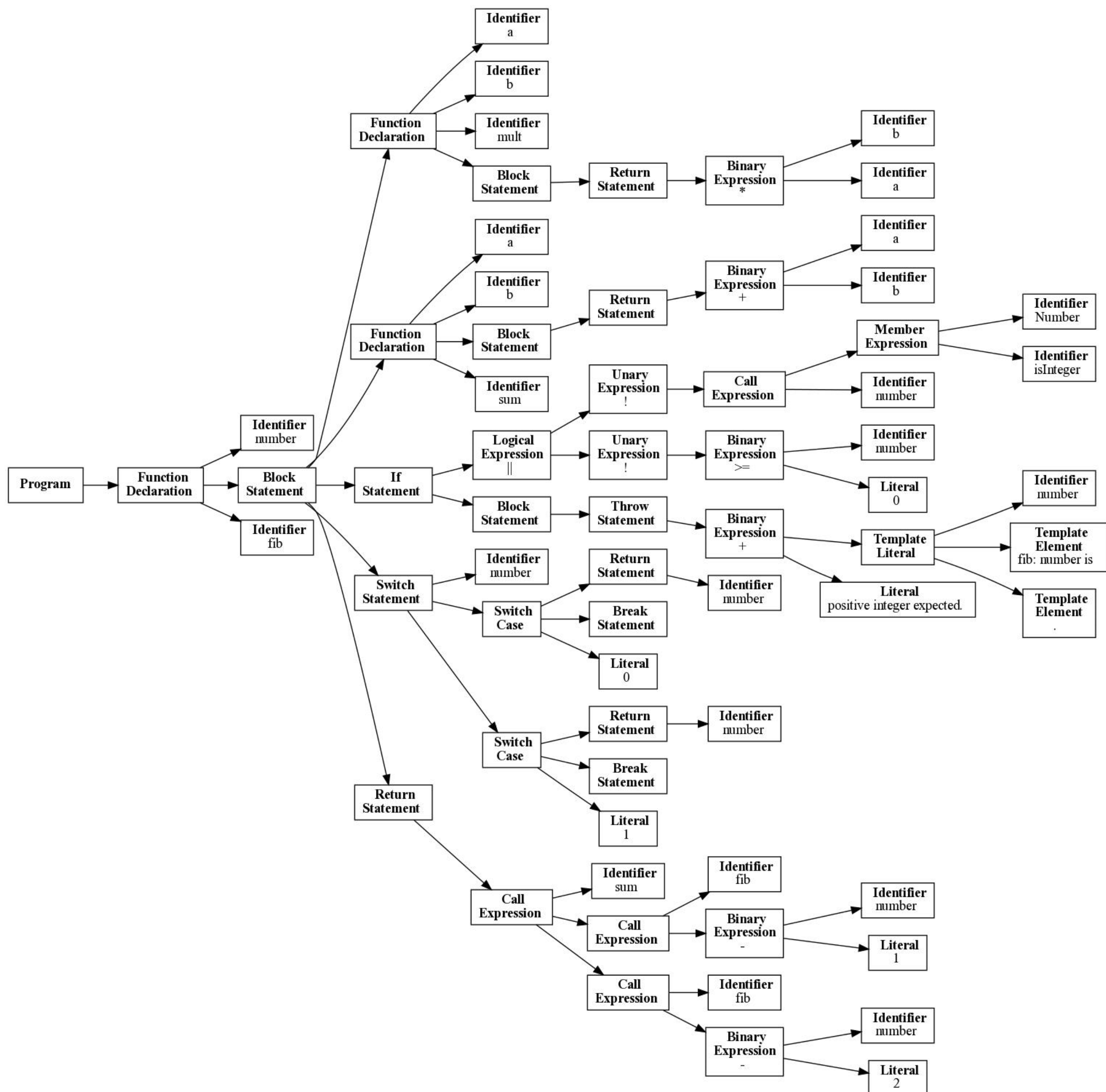

Figure 5: AST for original code, as shown in Figure 2, parsed by acorn

While the AST in Figure 5 seems rather large for such a short program, the AST for the minified version in Figure 6 looks far less wide. At tree depth level three, the AST in Figure 5 has five nodes, two of which are the function declarations of *sum* and *mult* in the original code. The AST in Figure 6 only has four nodes at depth level 3, omitting the two function declaration nodes but adding a variable declaration node. The branch of the switch statement remains in both variants but contains four nodes less in the minified version. For the subtree of the return statement, the next node is no longer a call expression but a sequence expression, spanning an additional six nodes. It is clear from comparing Figure 5 and Figure 6 that the minification has fundamentally changed how the code is represented by AST.

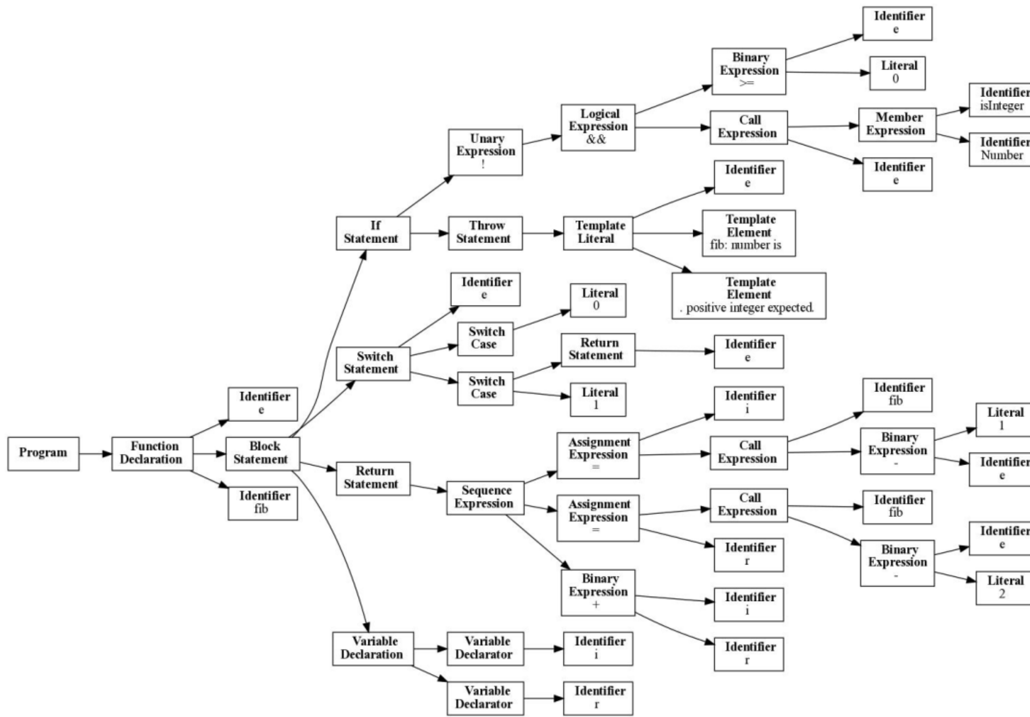

Figure 6: AST for minified code, as shown in Figure 3, parsed by acorn

During obfuscation the technique of control flow flattening has been applied, resulting in changes in the AST. The resulting AST is depicted in Appendix D, closer inspection is not required to see drastic changes: The amount of nodes is almost triple the number of nodes shown in the AST of the original non-obfuscated source code, as displayed in Figure 5. For more radical obfuscation techniques, i.e. dead code injection, the explosion of AST size is even more drastic. Enabling dead code injection additionally to the previously performed obfuscation techniques has resulted in an AST containing over 23 times the number of nodes of the AST corresponding to the unmodified source code.

For further analysis purposes, a large AST can pose a problem because the relevance of its individual parts is unclear, e.g. dead code or framework boilerplate are of less interest than use-case dependent custom features. One way to heuristically narrow down the code or AST is explained in Section 1.1.3 i.e. TF-IDF.

1.1.3 Code Similarity and Code Clones

Code similarity measurement employs many techniques that are known from text similarity measurement techniques for natural languages, for a survey on text similarity measurement and more in-depth information see J. Wang and Dong (2020).

Term Frequency - Inverse Document Frequency (TF-IDF) is a rather simple, but widely used statistical method to evaluate how important or common a term is for a document relative to a corpus. In the field of linguistics, a corpus is a written collection of naturally occurring language text, characterizing the state or variety of a language (Sinclair, 1991), for the context of this thesis the definition is extended to include published source code written in a programming language, specifically JavaScript.

The product of the term frequency (TF)

$$TF = \frac{number\ of\ occurrences\ of\ term\ in\ document}{number\ of\ terms\ in\ document}$$

and the Inverse Document Frequency (IDF)

$$IDF = \log\left(\frac{number\ of\ documents\ in\ corpus}{number\ of\ documents\ containing\ term\ in\ corpus}\right)$$

is the Term Frequency - Inverse Document Frequency

$$\mathrm{TF-IDF} = TF * IDF.\ \text{(Karabiber)}$$

Plainly, if the term occurs often in a document, but rarely in the corpus, it is considered important to the document. If a term occurs often in a document but also very often in the corpus, the term is unlikely to convey important information in the document, it could just be a common word in the language of the corpus. TF-IDF is often used for automatic keyword extraction. Amur et al. (2023) identify *KeyBERT*, *YAKE* and *RAKE* as current techniques for keyword extraction, with both *KeyBERT* and *YAKE* utilizing TF-IDF.

Code clones can simply be defined as a fragment of source code that is identical or similar to another piece of code by some similarity measure.
Code clones can be represented by four categories (Ain, Butt, Anwar, Azam, & Maqbool, 2019; Alfageh, Alhakami, Baz, Alanazi, & Alsubait, 2020; Runwal & Waghmare, 2017):

- **Type-1:** The code is identical except for small changes like whitespace, layout and comments.
- **Type-2:** The structure of the code is almost identical, identifiers' names, types, white spaces, and comments may be modified.
- **Type-3:** For the so-called near-miss clones, the modifications of Type-2 apply, additionally parts of the code can be deleted, or new parts added.
- **Type-4:** For semantic clones or logical clones, the functionality is identical, but the source code differs syntactically.

Relating the four categories of code clones to the typical code transformations performed on client-side JavaScript source code, as presented in Section 1.1.2, some observations can be made:

Type-1 and Type-2 clone types could occur by direct reuse of code, making small manual changes to fit the context of the application better or by formatting performed by linters. If the modifications are a bit heavier and include techniques such as tree shaking, they could also result in Type-3 clones. Depending on the aggressiveness of a minifier, the resulting code could be considered a Type-2 or Type-3 clone. Obfuscation, however, changes the syntax of the source codes drastically, resulting in a Type-4 code clone.

1.2 Motivation

Software Bill of Materials (SBOM) became of strong interest when the US declared the Executive Order on Improving the Nation's Cybersecurity (2021), stating that software vendors to federal agencies are required to provide an SBOM.

An SBOM is a document providing comprehensive information on the components and dependencies of a software project, this includes all components, their respective versions, the license information of each component and the interdependencies of the components. Multi-

ple specifications of SBOMs coexist and vary in the additional information they contain. CycloneDX (OWASP Foundation) and SPDX (The Linux Foundation) are some of the most popular standards.

The main benefits of SBOM consumption identified by a worldwide survey of technology professionals in 2021 (Hendrick, 2022) include the following:

- Security: Awareness of risky components and timely recognition of vulnerabilities
- Maintenance: Proactive recognition of components that reach the end of life
- Legal: Providing information about components relevant to reporting and compliance requirements
- Risk management: providing information addressing compliance, financial and reputational risks for leveraging third-party software

The survey also found that 98% of surveyed organizations use open-source software.

There is already a variety of software available to generate SBOMs during a build process using artifacts etc, for popular tools see Sham (2023). The core task of creating an SBOM for a software project is identifying the used components and their versions. Modern build processes for web applications utilize package management tools, such as yarn ("yarn," 2016) or NPM (npm, Inc., 2009), that are configured via a file, i.e. package.json, containing all the information on dependencies and their versions. This file is not deployed with the website, so this information is only privy to those with access to the development system, but not to the visitors of a website.

An SBOM can be a useful tool to securly maintain an application, but studies conducted in the last decade make harsh observations on the lack of website security and the negligent maintenance of remote inclusions:

Nikiforakis et al. (2012) conducted a large-scale web crawl to investigate the evolution of JavaScript inclusions over time. Developers have two options to include external libraries; by downloading a copy of the library and uploading it to their own webserver or by instructing the users' browser to fetch it directly from the third-party server. Nikiforakis et al. regard the former as a safer choice, as it gives the developer control over the website's integrity and the code that will be served to users. This comes with the drawback of higher maintenance costs, the potentially degraded website performance, as users may be forced to download a script that might have already been cached by their browser, and the potential ineffectiveness of it if the library loads additional remotely hosted code at runtime. Their study revealed that 88.45% of the crawled pages included at least one remote JavaScript library. They also tracked updates to the top 1k included scripts and found that roughly ten percent of scripts were modified at least once in a one-week period, resulting in their recommendation of conducting a weekly manual inspection of changes to the included scripts to increase the security of the script-including website.

Lauinger et al. (2017) conducted a study on client-side JavaScript usage and the resulting security implications. They inspected the Alexa Top 75k as well as a random sample of 75k websites and discovered that 37% of the inspected websites include at least one library with a known vulnerability and nearly 10% include two or more different vulnerable versions. They emphasize the need for more thorough approaches to dependency management, code maintenance and third-party code inclusions on the web, as the inclusion of vulnerable libraries poses a threat to the security of the website itself. While it is important to update vulnerable libraries in a timely manner, they discovered that many sites rely on libraries that are no longer maintained and that 61.4% of the inspected websites were at least one patch version behind on at least one of their included libraries. Even more surprisingly they discovered that it is not a rare phenomenon to include the same library – the same version or even multiple

different versions – in the same document, which can cause potentially non-deterministic behaviour with regard to vulnerabilities.

The top ten list of client-side security risks compiled by the OWASP Foundation (2021) included the use of vulnerable and outdated JavaScript components, as well as JavaScript drift, i.e. the undetected – possibly malicious – changes of script behaviour that could be introduced by third-party libraries.

These findings raise the question of whether those negligently maintained websites are relying on modern build and deployment processes, making use of the functionality provided by package management tools, or if they run on legacy code – possibly too difficult or cumbersome to reverse engineer and replicate for better maintenance. Software component analysis could help, but analyzing client-side code is a complicated task: The ever-changing ECMAScript specifications lead to tools requiring constant adaption or facing becoming obsolete quickly. The browser environment is a difficult terrain, HTML, and therefore scripts, are split into different iframes, and techniques like HOTDOL (Han, Ryu, Cha, & Choi, 2014) are applied to make it harder to gather the required information.

The dynamic nature of JavaScript, often dynamically loading more code at runtime, makes static analysis, and even just the task of downloading the client-side code, complicated.

Nikiforakis et al. (2012) noted that for their survey they had no success gathering the scripts by simply performing HTTP requests, they applied a more sophisticated approach of using a headless browser pretending to be Mozilla Firefox 3.6. This allowed them to execute the inline JavaScript code and resolve remote script inclusions, at least the ones intended for Mozilla Firefox's browser.

After collecting the scripts of a website, the analysis of the source code is still quite a difficult task: Client-side code is often modified to increase the performance of a website, resulting in hardly human-readable code. Common transformations include minification, sometimes called uglification, obfuscation or bundling of multiple source code files. Also, developers often make adjustments to their local copies of third-party components to better fit their needs (Godfrey & Kapser, 2021). In addition to JavaScript, there is also *WebAssembly* (Rossberg, 2019), an increasingly popular compilation target for many languages that can run alongside JavaScript in the browser.

Paired with fast-evolving libraries in a fragmented JavaScript ecosystem and the lack of tool support for JavaScript, this makes the identification of third-party libraries on websites a very difficult task.

But why would this be interesting, after the website has already been deployed?

Mistakes happen easily, especially when complex build processes, pipelines and tools, e.g. *webpack* (Koppers et al., 2016), with a myriad of optimization settings, are involved. Unused code could be optimized away, but also source code comments containing license information, i.e. license identifiers or even full license texts, could – unknowingly – be removed for optimization purposes, as demonstrated in Section 1.1.2 . Additional code could also be introduced, which is especially dangerous when GenAI tools like AI website builders or AI content creators are used, as this can cause accidental copyright implications (Eckhardt & LL.M. Lüttel; Endres & Mühleis).

Creating an additional SBOM after the deployment of the websites could therefore help shed light on misconfigured build/deployment processes as well as help remedy potential security risks and license compliance infringements faster.

The creation of an SBOM for already deployed websites could also be immensely useful for maintainers of legacy websites or web applications that do not have the dependency information readily available.

1.3 Objective

This thesis aims to explore the feasibility of constructing an SBOM for the client-side scripts of any given website. The difficulty in this lies in the nature of client-side scripts, which are often modified, minified, obfuscated and collated by bundlers – so the focus of this thesis will be on the identification of client-side JavaScript libraries and their versions.

To identify a suitable tool, research on clone detection tools and mechanisms for JavaScript has been conducted and the tool *Siamese* (Ragkhitwetsagul & Krinke, 2019) has been selected for closer examination. A corpus containing popular JavaScript libraries is created, which is then transformed by leveraging the tool *Siamese* to create multiple representations to query libraries to be identified against. A proto-benchmark is constructed of JavaScript libraries in their original form and in modified forms, transformed by various modifications that are typically applied to client-side scripts. The suitability of *Siamese* to identify JavaScript libraries using its code clone detection mechanism and its discriminative power is evaluated using the proto-benchmark.

1.4 Thesis Structure

In the first section, 1.1 the necessary fundamental knowledge of JavaScript, client-side code transformations, i.e. minification, obfuscation, transpilation and bundling, code similarity measures and code clones are laid out. The motivation to identify the libraries and their version, and ultimately build an SBOM, for client-side code has been stated in Section 1.2 The objective is formulated in Section 1.3

In the next chapter, a thorough literature review on code similarity and identification of code clones, specifically for JavaScript, will be conducted. In Chapter 3 the functional and non-functional requirements for the aspired tool are defined.

In Chapter 4 the architecture for the chosen approach is outlined, followed by the explanation of the design and implementation in Chapter 5 . First, the underlying assumptions are explicitly stated, and then the resulting design choices are discussed. Section 5.3 is focused on utilizing the tool *Siamese* and the resulting difficulties, while Section 5.4 is focused on the creation of a benchmark for the evaluation of the tool.

In Chapter 6 the results achieved on the benchmark are presented and discussed, and then the system is evaluated against the requirements previously formulated in Chapter 3 .

Lastly, in Chapter 7 the conclusion is presented, as well as possible threats to validity and an outline of future work.

2 Literature Review

Measuring the similarity of source code comes in many application forms, such as plagiarism detection, malware analysis, clone or reuse identification as well as code recommendations, e.g. such as detection of code smells. For the purpose of this thesis, the domains of plagiarism detection as well as code clone detection have been reviewed.

Interesting techniques for plagiarism detection in source code include *Winnowing* (Schleimer, Wilkerson, & Aiken, 2003), which is used by the tool *Moss* (Aiken), and *Extended-Winnowing* (Shrestha, Shakya, & Gautam, 2023). Both algorithms employ hashing and fingerprinting to identify similar code segments. *Winnowing* is insensitive towards whitespace, suppresses noise and is position independent, but what sets it apart is the guarantee to detect at least one k-gram in any shared substring of $length \geq window\ size + k - 1$ due to the selection of fingerprints.
Devore-McDonald and Berger (2020) explored evading plagiarism detectors by performing semantics-preserving code modifications, proving that popular plagiarism tools like *Moss* (Aiken), *Sherlock* (Pike & Loki), *JPlag* (Mahlpohl) and *FETT* (Nichols, Dewey, Emre, Chen, & Hardekopf, 2019) are not robust against their modifications or code obfuscation.
The suitability of the technique *Winnowing* for clone search is further explored in Section 4.1 by analysing code clones using *Moss*.

Jensen, Madsen, and Møller (2011) propose a tool for static analysis of JavaScript (ES3) and highlight the difficulties resulting from the dynamic nature of JavaScript as well as the interactions with the HTML DOM and browser API. While some of their work on collecting the scripts from websites might still be relevant, their tool for static analysis is far too outdated for modern JavaScript source code.

Walker, Cerny, and Song (2020) conducted a survey on code clone detection tools in the timeframe of 2009 until 2019. They found that the ratio of not publicly available/closed-source tools vs open-source tools was nearly 4:1. They also found that while a wide range of languages is covered by modern clone-detection tools, the by far most popular language for clone-detection tools is Java, followed by C and C++. They speculate that this could be caused by the lack of standardized benchmarks for other languages, as the most popular benchmarks, Bellon's (Bellon, Koschke, Antoniol, Krinke, & Merlo, 2007) and BigCloneBench (Svajlenko, Islam, Keivanloo, Roy, & Mia, 2014), only cover C and Java.

Alfageh et al. (2020) conducted a literature review on code clone detection techniques for JavaScript as well as language-independent research. As tools for JavaScript, they identify *JSCD* (Cheung, Ryu, & Kim, 2016), *DECKARD* (Jiang, Misherghi, Su, & Glondu, 2007), *JSInspect* (St. Jules, 2014) and JSCPD (Kucherenko, 2019).
DECKARD calculates a vector representation of the AST of source code and applies a variation of Locality Sensitive Hashing (LSH) to cluster similar vectors, i.e. potential code clones. *JSCD* is based on *DECKARD* with the main difference of using *SAFE* (Lee, Won, Jin, Cho, & Ryu) to generate the AST for JavaScript code.
JSInspect is a node CLI tool supporting ES6, JSX and Flow. It generates an AST and compares node types based on the AST, the threshold for similarity is to be set by the user.
JSCPD is a CLI tool using hashing and the Karp and Rabin algorithm (Karp & Rabin, 1987) to identify copy-pasted code.

Zakeri-Nasrabadi, Parsa, Ramezani, Roy, and Ekhtiarzadeh (2023) conducted a comprehensive literature review on source code similarity measurement techniques. They categorize approaches as graph-based, tree-based, text-based, token-based, learning-based, metric-based or hybrid techniques.
While they identify 80 software tools for code clone identification, the majority of tools do not support JavaScript. The tools they identified with support for JavaScript include *LICCA* (Vislavski, Rakic, Cardozo, & Budimac, 2018) and *SourcererCC* (Sajnani, Saini, Svajlenko, Roy, & Lopes, 2015).
LICCA is a token-based and tree-based tool for cross-language clone detection. It creates a tree-based intermediate representation of code and applies a variant of the Longest Common Subsequence (LCS) algorithm for clone detection. *LICCA* is not robust against functionally similar fragments with syntactically different representations, e.g. loops and recursion. Sadly the repository is no longer accessible on GitHub, so it was not possible to confirm if this limitation still applies or perform any experiments using the tool.
SourcererCC is a token-based near-miss clone detection tool, as a similarity measure it calculates the overlap of source tokens among code blocks. By employing filtering heuristics to reduce the number of necessary candidate comparisons it improves its execution time and scalability. *SourcererCC* is not able to detect Type-4 clones.

Lončarević, Skendrović, Kovačević, and Groš (2023) offer a static detection approach for JavaScript libraries based on comparing cryptographic hashes and identifiers. This approach can only tolerate minimal modifications to the source code.
Pagon, Skendrovć, Kovačević, and Groš (2023) performed static code analysis to identify the version of an unknown JavaScript library by first identifying the library and then comparing the cryptographic hashes of different versions. De-minification was performed using *JSBeautifier* (Einar & Newman) to adapt to minified libraries and reduce the problem to a canonical form.

Ragkhitwetsagul and Krinke (2019) created the code clone search tool *Siamese* which creates multiple code representations by extracting lexical information using an *ANother Tool for Language Recognition* (ANTLR)-generated parser and lexer (Parr & et al) and an ANTLR Grammar (Kiers & Kochurkin), extracting methods and tokens, then performing normalization and n-gram generation.
Misu and Satter (2022) conducted an exploratory study using *Siamese* to analyse JavaScript code snippets on StackOverflow and GitHub repositories for code clones.
Siamese is further examined in Chapter 4 and ultimately used as the tool for clone search in this thesis.

Cao, Peng, Jiang, Falleri, and Blanc (2017) offer an approach to automatically browse web applications and retrieve the client-side JavaScript libraries used by the website. Their goal was to identify the libraries used on popular websites as a guide for developers faced with the choice of which libraries and versions to use. They use a combination of three strategies to identify libraries and their versions; they check if the code comments contain any library name, they compare the source code against reference files and they execute a sensor JavaScript plugin to dynamically identify the usage of libraries.

The tool *Retire.js* (Oftedal, 2018) is available as a node CLI scanner, grunt plugin, gulp task, browser extension and burp extension. It is able to identify several JavaScript libraries with known vulnerabilities and can generate an SBOM of the detected libraries. The libraries that are enabled for detection need to be specified in a JSON file, containing information for each library with regards to their known vulnerabilities as well as extractors, e.g. the regular expressions to identify the filename and the name of the library in the file content, and an

optional mapping between hashes of file content and versions. Further investigation in Section 4.1 revealed that the construction and extension of this JSON file seem to be a manual and rather tedious task and therefore the approach is deemed not suitable to be extended as a general-purpose library identification tool.
Lauinger et al. (2017) performed a study on JavaScript library inclusions on the web, using both a dynamic approach similar to *Retire.js* and a static approach by comparing file hashes. They also experimented with a simple “name-in-URL” heuristic, which led to a large number of false positives. Searching for the library name “jquery” resulted in plenty of files not containing the code of the library but plugins for *jquery*. Other files contained additional code or libraries and a large number of files could not be identified by this heuristic because the files were renamed by developers.

X. Liu and Ziarek (2023) implemented *PTDECTOR*, a browser extension for Chrome to identify front-end libraries on websites, even when they are bundled by packers like *webpack* (Koppers et al., 2016). The tool uses JavaScript library files and their dependencies as input and constructs a data structure called pTree, modelling the libraries' properties, which are then compared by a weight-based tree-matching algorithm. The tool is not able to detect the versions of libraries and is not able to work with ES6 modules that allow partial library loading. The required dependency information is collected manually, making the database for the tool in its current form hard to extend. Browser extensions with similar functionality include *Library Detector* (Bredow & Michel, 2010), *Library Sniffer* (justjavac, 2020) and *Wappalyzer* (Wappalyzer Pty Ltd, 2023a). The Wappalyzer browser extension is examined more thoroughly in Section 4.1 but cannot be considered a candidate for the clone search tool of this thesis because of its proprietary character.

Hammad, Babur, Basit, and van den Brand (2022) present *Clone-Seeker*, a code clone detector that allows search queries based on source code or natural language. For this approach, a natural language document of metadata is generated for each code clone. The objective of allowing natural language queries is the improvement of code search on websites like GitHub or StackOverflow. Although the work of these authors cannot be directly used in this thesis, it shows beautifully how vast the research field and its applications are, demonstrating that code clones can exist across the programming language and natural language divide.

Huang et al. (2023) studied the effect of obfuscation on clone detection techniques on Java code, specifically on code clones undergoing modification before obfuscation. They discuss how traditional clone detection techniques and deep learning-based techniques are differently affected by various types of obfuscation: On traditional techniques control flow obfuscation seems to have the most severe impact, while this also impacts deep learning-based detectors it does so less severely. Notably, deep learning-based detectors are quite affected by the simple obfuscation technique of identifier replacement. While it is unclear how well their results translate to clone detection on JavaScript source code, their work accentuates the need to include obfuscated source code in a benchmark when testing a clone detection tool.
For languages with more tool support, i.e. Java, a variety of other approaches exist:

Analysing the bytecode for code clone identification (Akram, Mumtaz, & Luo, 2020; Wan, Dong, Zhou, & Qian, 2023; D. Yu, Yang, Chen, & Chen, 2019), deep learning-based approaches leveraging pre-trained models such as *CodeBERT* (Arshad, Abid, & Shamail, 2022; Feng et al., 2020) or approaches based on program dependency graphs (PDGs) (Zou, Ban, Xue, & Xu, 2020).

Recently, more extensive benchmarks (Alam et al., 2023; Al-Omari, Roy, & Chen, 2020) for clone detection have been created, sadly not including JavaScript code snippets.

T. Wang, Harman, Jia, and Krinke (2013) introduced *EvaClone*, a tool offering a search-based solution to identifying the optimal configuration parameters, i.e.the confounding configuration choice problem, of clone search tools, also enabling a more meaningful comparison of tools. A byproduct of this work is the General Clone Format (GCF) and the GFC converter, making it possible to convert the outputs of multiple clone search tools into the same format for better evaluation. One of the formats convertible to GFC is the Rich Clone Format (RCF), proposed by Harder and Göde (2011).

From the literature review performed in this chapter, it has become quite obvious that there is a need to create a benchmark for clone detection on JavaScript source code. Unfortunately, it has also become clear that tools built for JavaScript do not age well, as the continuing evolution of ECMAScript specifications quickly renders previously working tools useless rather fast. Many tools seem to be no longer maintained, possibly because it would have been too much work to adapt them to the newly added language features.

3 Requirements

At first, the scope of the thesis started out much wider, hence the name *URL2SBOM*, so the requirements set out for this thesis evolved a lot during the process of writing the thesis.

In the following sections, the functional and non-functional requirements and their evolution will be defined and explained. The requirements defined in this chapter will be used for evaluation in Chapter 6

3.1 Functional Requirements

At first, the scope of the thesis included the functional requirements of identifying unknown JavaScript libraries and their version from any given website and to create an SBOM.

This scope turned out to be too wide given the current state of research on clone search for JavaScript, the focus of the thesis changed towards investigating the feasibility of identifying an unknown JavaScript library after transformations typical for websites have been performed.

The following functional requirements reflect this new narrowed-down scope:

Corpus:

F-01 The corpus should only contain JavaScript source code for client-side execution.

F-02 The corpus should contain a representative selection of JavaScript libraries.

F-03 The corpus should contain source code covering multiple ECMAScript versions.

F-04 The corpus should contain all relevant versions of a library.

Corpus Builder:

F-05 The corpus builder should allow for easy extension of the corpus.

Tool for Clone Search:

F-06 The selected tool needs to be able to process and index a representative corpus of JavaScript client-side libraries.

F-07 The selected tool needs to produce human-readable output.

F-08 The selected tool needs to take a config file as input to enable flexible settings.

F-09 The selected tool needs to be able to detect code clones robustly on <u>minified</u> source code.

F-10 The selected tool needs to be able to detect code clones robustly on <u>obfuscated</u> source code.

F-11 The selected tool needs to be able to detect code clones robustly on <u>bundled</u> source code.

F-12 The selected tool needs to be able to detect code clones robustly on transpiled source code.

F-13 The selected tool needs to be able to detect code clones robustly on modified source code.

JavaScript Benchmark:

F-14 The benchmark should contain JavaScript client-side source code in various ECMAScript versions.

F-15 The benchmark should contain JavaScript client-side source code transpiled to ES5.

F-16 The benchmark should contain modified JavaScript client-side source code.

F-17 The benchmark should contain minified JavaScript client-side source code.

F-18 The benchmark should contain obfuscated JavaScript client-side source code.

F-19 The benchmark should contain bundled JavaScript client-side source code.

3.2 Non-Functional Requirements

The aforementioned wider scope at the beginning of this thesis included the non-functional requirements of writing the *URL2SBOM* tool in Java using the Spring Boot framework and providing the application wrapped in a Docker container.

The following non-functional requirements reflect the current scope:

Corpus Builder:

NF-01 The corpus builder application should be written in Java.

NF-02 The corpus builder application should be efficient, using parallelization and avoiding unnecessary network traffic.

Tool for Clone Search:

NF-03 The selected tool should be licensed under an open-source license.

4 Architecture

In this chapter, the research conclusions will be presented and based on said research conclusions and further experimentations a suitable tool for clone search on JavaScript client-side code will be identified. To anticipate the next section, the selected tool is *Siamese*. The choice of the tool *Siamese* will be motivated and its architecture will be explained in Section 4.2 .

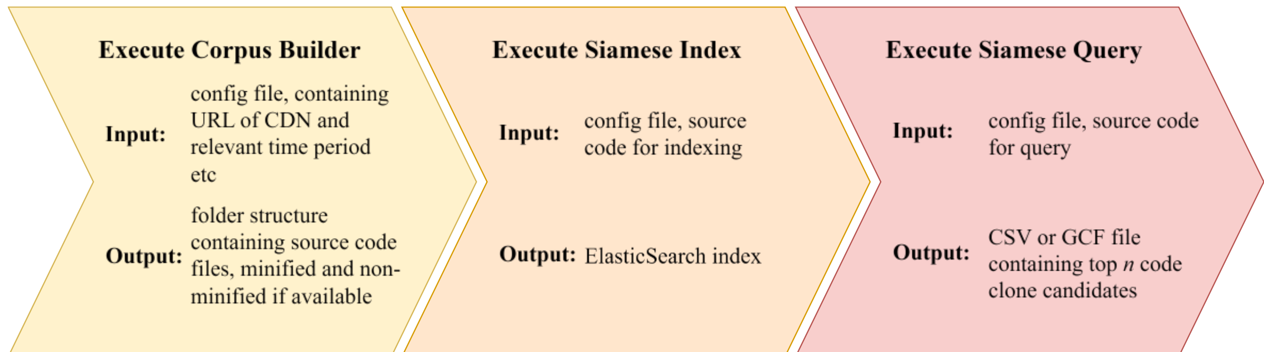

Figure 7: Combining the corpus builder and Siamese for clone search

As shown in Figure 7, the corpus builder – described in Section 4.3 – is created with the purpose of downloading JavaScript client-side code that can be used as input for indexing by *Siamese*. Executing *Siamese* for indexing creates a searchable *ElasticSearch* index which then can be used to execute queries with *Siamese* to identify unknown libraries. *ElasticSearch* is a distributed RESTful search and analytics engine, suitable for analysis of large datasets and full-text search (Bannon & et al, 2010).

4.1 Research Conclusions

The literature review, see Chapter 2 , indicates the need for a benchmark for clone detection on JavaScript source code. Unfortunately, tools built for JavaScript do not age well, as the continual evolution of ECMAScript specifications quickly renders previously working tools useless. In many cases, tools are no longer maintained, possibly due to the amount of work needed to adapt them to new language features. For the purpose of this thesis, three different approaches found in the literature were investigated.

The first approach was utilizing *winnowing* (Schleimer et al., 2003), as is done by the software similarity detection tool *Moss* (Aiken). To investigate this approach, the required account was obtained and *Moss* itself was used to compare a small sample set of files. The output of *Moss* comparing two obfuscated variations of the code disclosed in Figure 2 is shown in Figure 8. Even though the functionality of the obfuscated code has not changed, the syntactical representation looks quite different, making it a Type-4 code clone. *Moss* was only able to identify 2% of the heavily obfuscated code and 19% of the lightly obfuscated code as potential code clones. Comparing the source code of *jquery@3.7.1* with its minified version led to only 12% of the non-minified and 20% of the minified code being recognized as a potential code clone. *Moss* offers some insight into its matching by providing the lines of code that it identified as similar, sadly this does not translate well for code without line breaks, i.e. code that was minified or heavily obfuscated. For other seemingly different libraries, such as *lethargy@1.0.2* and *locomotive-scroll@3.0.0*, *Moss* found that 80% of lines in *lethargy@1.0.2* are candidates for code clones. This shows that transformation techniques like minification and obfuscation lead to lower *Moss* scores than randomly paired libraries, making setting a sensible threshold, that accounts for both transformed and regular source code, impossible.

The idea of following this approach and implementing *winnowing* was therefore discarded, as it turned out not to be robust enough against source code minification or obfuscation.

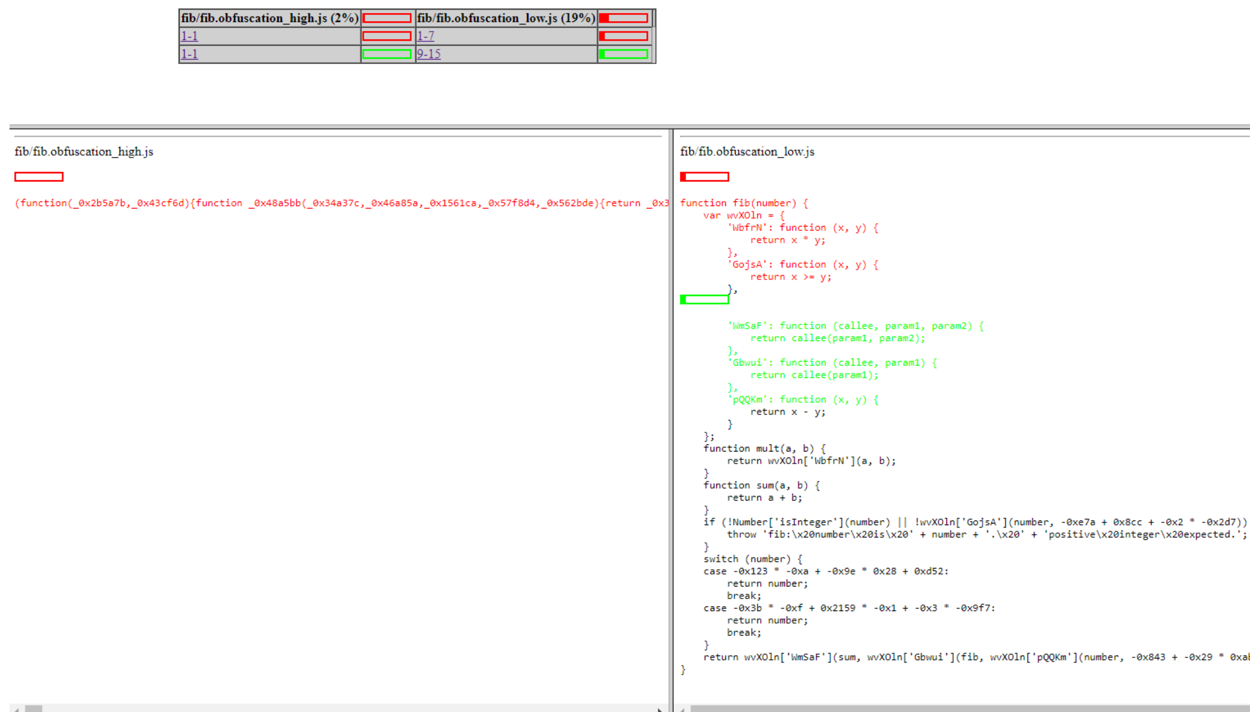

Figure 8: Using Moss to compare two obfuscated variations of the same code snippet

The second approach was utilizing a browser extension performing dynamic analysis of websites and possibly extend their corpus as well as their detection technique. The tool *Retire.js* (Oftedal, 2018) was investigated more thoroughly, revealing its corpus only contains information on libraries and their versions with known vulnerabilities, less than 500 libraries in total. Extending the corpus to be able to recognize a wider variety of libraries seems to involve a lot of manual work, as unique extractors have to be defined for each library. While investing time and effort to extend this corpus is possible, the corpus would not stay relevant for long and the associated cost of maintaining it is simply unreasonable.

This is not to say that it cannot be done, as the proprietary Wappalyzer browser extension (Wappalyzer Pty Ltd, 2023a) offers similar functionality as Retire.js but comprises a knowledge base spanning over 4k libraries and other web technologies. Their detection mechanisms are also more sophisticated, leveraging more information such as an implies-relationship, where the detection of one library automatically implies the detection of another library that is required for the detected library to function, e.g. *jPlayer* implies *jQuery* to be present. They also use typical DOM elements to identify libraries, e.g., links, style elements, images, iframes etc. containing a characteristic link to the library or technology. For some technologies, the headers of HTTP requests or cookies contain characteristic data used for identification. For JavaScript libraries a command is dynamically injected and evaluated, prompting the library and its version as output. Using multiple detection mechanisms and a large knowledge base, the browser extension is able to analyze thousands of websites and gather data on the technologies used. This seems to be Wappalyzer's core business model, analyzing websites and selling data in different representations, therefore the effort and cost of curating and maintaining such a knowledge base might be profitable for them.

The third approach was using the tool *Siamese*. It was already used to conduct research successfully on JavaScript (see Misu & Satter, 2022) and seemed rather adaptable, considering ANTLR grammars are easily updated for newer versions and can be manually modified as well. The maintenance of a code corpus seems very straightforward, as code can be added incrementally without effect on previously added code. This makes *Siamese* an interesting candidate for further investigation. The rather sophisticated architecture of *Siamese* will be discussed in Section 4.2 .

4.2 Siamese

In this section the mechanisms of the tool *Siamese*, as presented by Ragkhitwetsagul and Krinke (2019), will be outlined.

Siamese takes a configuration file, as shown in Appendix A, as input, specifying the location of source code, whether to index or search, the size of n-grams etc. The output of the search command can be specified to be a CSV or a GCF file. As shown in Figure 10, the tool operates in two phases: Indexing and retrieval, with the indexing phase being rather long and computationally expensive due to the amount of data being processed.

In the indexing phase, a corpus of source code will be consumed to generate a searchable code index. In Figure 9, and throughout Ragkhitwetsagul and Krinke (2019), the corpus is referred to as a Java corpus, but for the sake of this thesis, a JavaScript corpus is used instead. The first step is to generate an AST, an intermediate representation of the source code, to extract methods and tokenize the code. *Siamese* leverages ANTLR to parse the code.

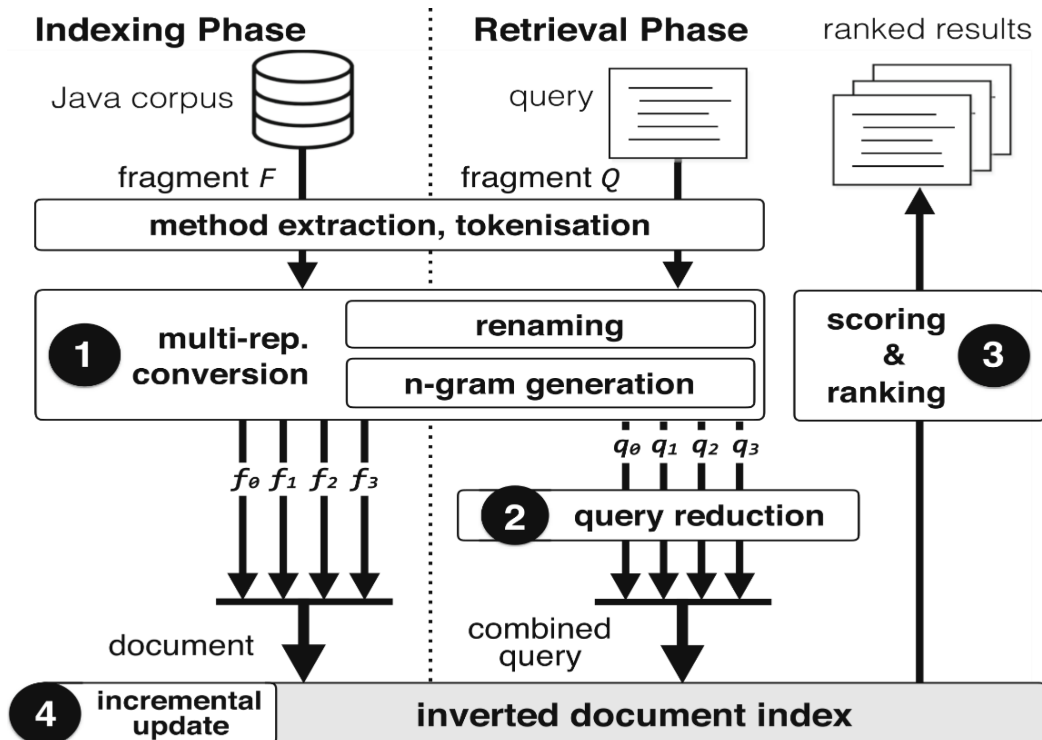

Figure 9: Siamese architecture. From Ragkhitwetsagul & Krinke, 2019

Next, the multi-representation module creates four code siblings, as depicted by Figure 10: For text search the original source file is saved as a stream of tokens or 1-grams, for the identification of Type-1 clones a stream of n-grams is generated, for Type-2 clones a stream of

partially normalized n-grams is generated, where identifier, literal and type tokens are replaced by normalized tokens, and for Type-3 clones a stream of fully normalized tokens is generated, where all tokens except Java punctuators are replaced. The four code siblings are then combined into one document, saving the generated representations in different fields.

Then the document will be added to the inverted document index, realized by ElasticSearch, via the incremental update module. These incremental updates allow users to add, delete or edit code fragments in the index without having to reindex the corpus again.

Siamese distinguishes between files and methods as code fragments, it also claims to be resilient to incomplete or uncompilable code fragments, just storing the source code at the file level.

In the retrieval phase, a piece of source code to be queried will go through a similar process as the indexing phase. After the extraction of methods and tokenization, the multi-representation module performs the same transformation, creating a combined query containing the four representations as subqueries.

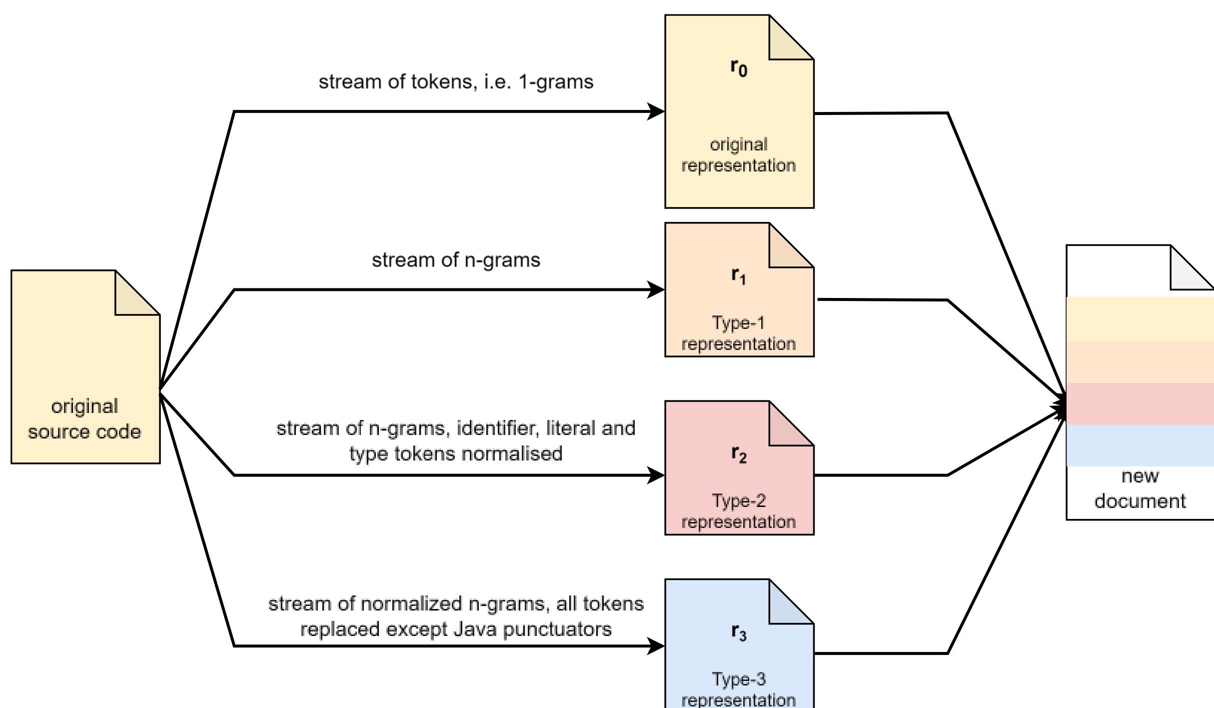

Figure 10: The four representations generated by Siamese

To avoid the “long query problem” (Kumaran & Carvalho, 2009), a query reduction technique is applied: The query is rewritten to only contain the rare tokens, by analyzing the tokens’ document frequency and comparing it to a threshold and discarding frequently occurring tokens. This narrows down the number of retrieved code snippets to highly relevant ones, speeds up the search and avoids false positive results.

Since *Siamese* is built upon *ElasticSearch*’s infrastructure, the scoring and ranking function of *Apache Lucene*, which is the backbone of *ElasticSearch*, is used. It represents code fragments as k-dimensional weight vectors, where k equals the number of terms in the dictionary, applying a variation of TF-IDF with some additional term boosting weights as a weighting scheme. A relevance score between the query vector and a document vector is calculated, and relevant documents are ranked according to their scores. To account for the four different code siblings, *Siamese* adds a custom scoring function, adding the scores of the siblings and returning the top n results to the user.

4.3 Corpus Builder

The goal of the tool described here is to construct a corpus of JavaScript source code by downloading information about available libraries, available versions and then the actual source code from a code registry or CDN, ideally downloading in parallel and avoiding downloading files multiple times to avoid unnecessary network traffic.

The source code is split into three packages, the download package as shown in Figure 11, the transform package shown in Figure 12 and the execute package shown in Figure 13. A simplified sequence diagram of the corpus builder application is displayed in Figure 14, the block *Download-Source-Code* is only depicted once instead of three times for simplicity.

The download package contains the classes *Reader*, *DownloadTrio* and *Downloader*.

The class *Reader* contains functionality to read files from disk into either a String or a Set of Strings. The class *DownloadTrio* defines a datatype for downloading files, containing the filename, the URI and a Future object of the file to be downloaded. The class *Downloader* implements Callable and is used to create instances of download tasks that are then passed to an ExecutorService for parallel execution.

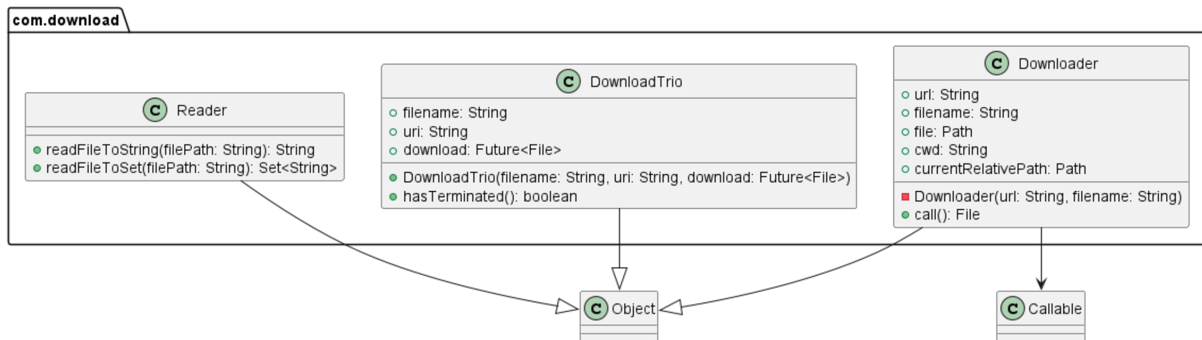

Figure 11: UML of the download package

The package *Transform* contains the classes *LibJSON* and *SaveToDisk*. tthe latter provides the functionality to save JSON files, the list of download jobs or source code to disk.

The class *LibJSON* provides the functionality to extract relevant information from downloaded JSON files, i.e. the available libraries' names, the versions available for a library, and the entry point for a library's version. It also contains a version filter, removing versions not suitable for production. The – experimental – functionality to extract non-minified source code from a source map file is also contained in this class.

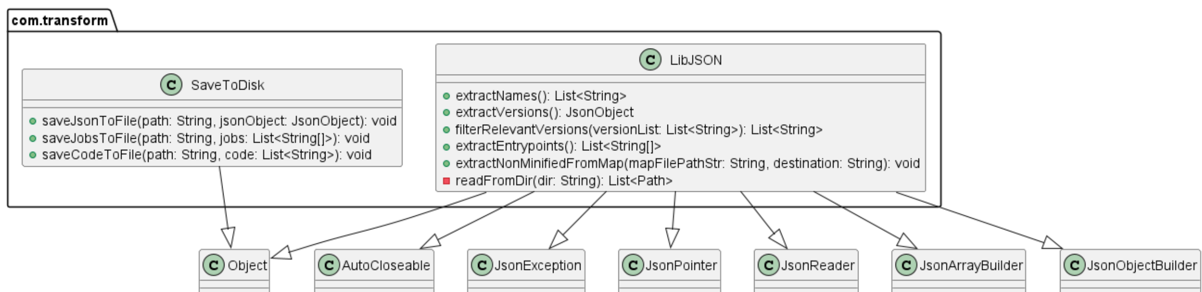

Figure 12: UML of the transform package

The package *Execute* contains the classes *config* and *App*, with *config* containing the user-defined variables for execution, including the maximum number of threads, URLs specific to the CDN, the destination where downloaded files should be saved, prefix for filenames etc.

The class *App* is the entry point of the actual execution, providing the main-method. The process of downloading the source code from the CDN contains multiple steps:

First, downloading a list of available libraries from CDN by submitting a *Downloader* task to an ExecutorService. Then utilize LibJSON to extract the library names from the downloaded JSON file. For each library, the information of available versions needs to be downloaded by submitting a *Downloader* task to the ExecutorService. For each downloaded file *LibJSON* is used to extract the available versions and filter them.

A dictionary containing the libraries names and the available versions is then saved as a JSON file, utilizing *SaveToDisk*. For each version of every library a file containing the URI to the entry point of said version needs to be downloaded, yet again by submitting *Downloader* tasks to the ExecutorService. The entry points are then extracted using the functionality provided by *LibJSON* and the resulting jobs, downloading the content located at every extracted URI, are then saved to disk using *SaveToDisk*. The downloading of source code is then started by submitting the jobs as *Downloader* tasks to the ExecutorService.

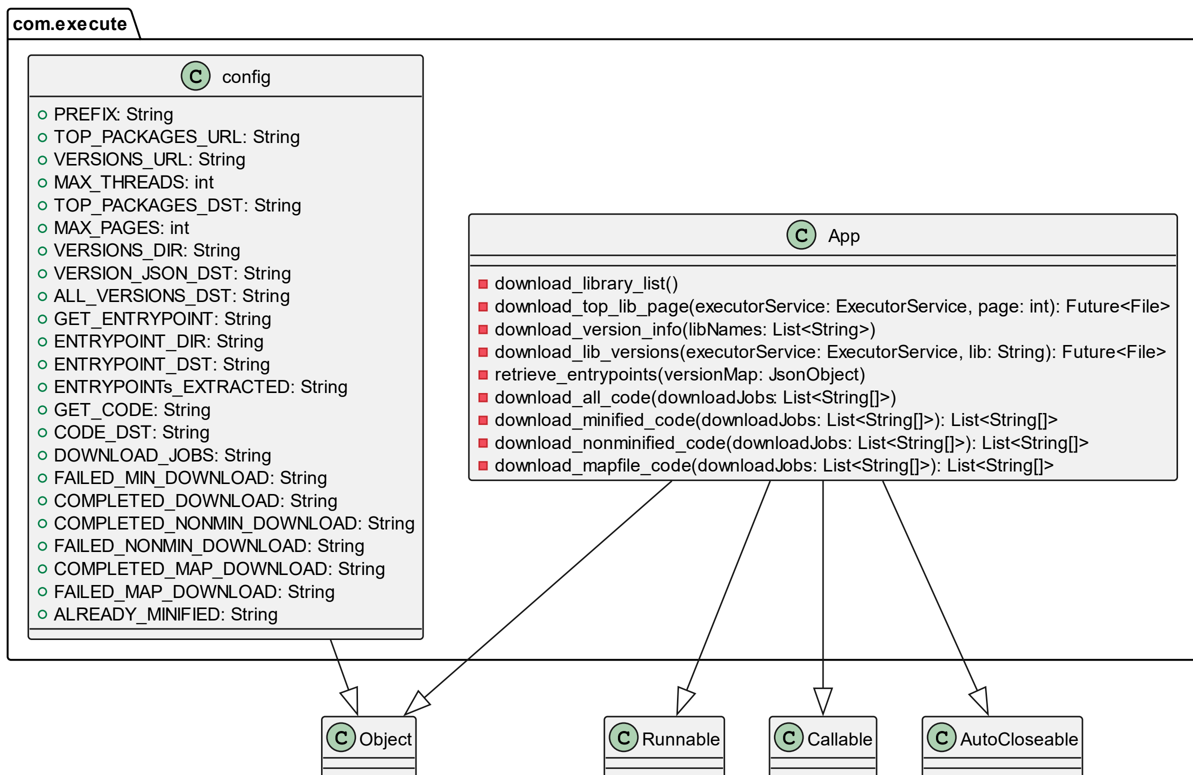

Figure 13: UML of the execute package

The saving of the dictionary containing the version information and the list of download jobs is not necessary for the actual process of downloading source code, it is done for practical reasons: Debugging and resuming the rather lengthy process at a different time.

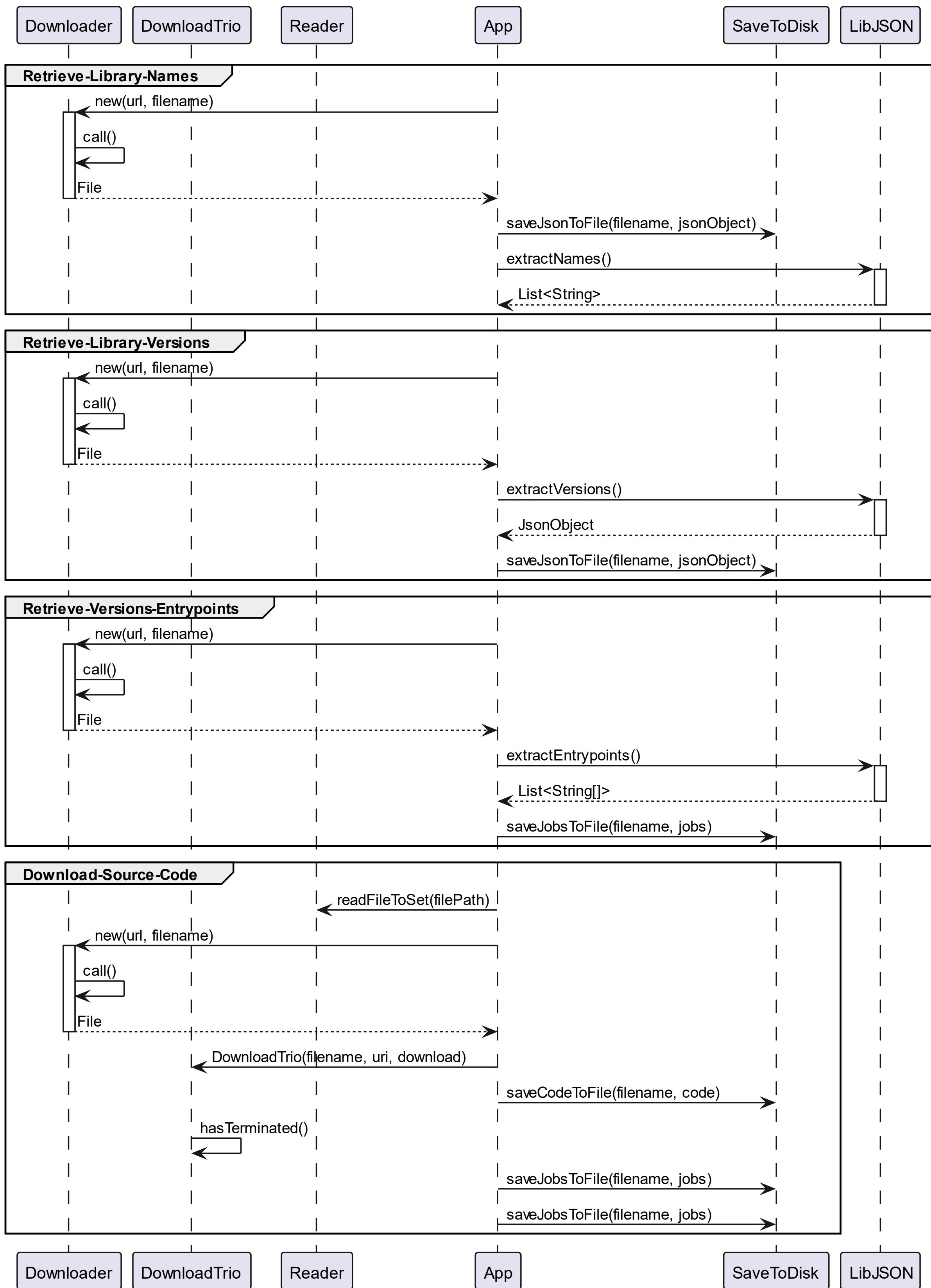

Figure 14: Simplified Sequence Diagram of the Corpus Builder Application

5 Design and Implementation

In this chapter, first, the design for the clone search architecture - including assumptions and their implications for the design – will be outlined in Section 5.1 The implementation of the corpus builder application will be discussed in Section 5.2 providing additional details to its architecture outline in Section 4.3 In Section 5.3 the adoption of *Siamese* for clone search will be detailed. The architecture of *Siamese* is sketched out in Section 4.2 for more details on its implementations please refer to Ragkhitwetsagul and Krinke (2019). Lastly, Section 5.4 investigates the construction of a benchmark for JavaScript clone search, defines a small proto-benchmark and provides the result achieved for it.

5.1 Design

While the architecture, as discussed in Chapter 4 is rather straightforward, the concrete design is influenced by many assumptions based on personal observations due to the lack of reliable sources and relevant literature on the ever-changing nature of the World Wide Web. The assumptions have been as explicitly laid out as possible in Section 5.1.1 and the implications for the design and implementation are discussed in Section 5.1.2

5.1.1 Assumptions

Assumption 1 The most popular sources for JavaScript code are GitHub and NPM, but both contain also code not relevant to client-side scripting, as well as unmaintained, unpopular and abandoned code.

Assumption 2 CDNs are a popular way to access packages available on NPM or GitHub, so they provide a reasonable metric on how relevant and popular code is for client-side execution.

Assumption 3 New versions of packages are published under the same name with a different version number, following the semantic versioning convention of Major.Minor.Patch (Preston-Werner).

Assumption 4 Versions of a package are similar and have some code in common.

Assumption 5 Versions of a package can be published under different licenses or may contain security issues only in certain versions, i.e. identifying the version of a package is relevant.

Assumption 6 Only versions that are suitable for production are used in production, i.e. versions containing the identifiers like release candidate, alpha, beta, experimental, test, prerelease etc. are not considered suitable for production – they are only considered if no suitable version of this library exists.

Assumption 7 Websites do not always use the latest version of a library, even if patches are available, i.e. outdated versions of libraries are still in use.

Assumption 8 Client-side code can be delivered in its original state, but also obfuscated, minified, modified, as a bundle or only in parts.

Assumption 9 In JavaScript applications, the entry point file is the first file to be executed. In this file, the application is typically set up, and configured, and may import other components or modules. This makes the entry point file the most relevant file for analysis.

Assumption 10 The names of packages do not contain characters not suitable for URIs, e.g. ‘~’, ‘\’, ‘|’, etc. While this is enforced by NPM (npm, Inc., 2024), it shall be assumed to

also be enforced by other platforms and that CDNs do not change the names of hosted libraries.

5.1.2 Resulting Design Choices

Based on *Assumption 1* and *Assumption 2* it is a sensible choice to use a CDN as a source for packages instead of obtaining them directly from either NPM or GitHub. For this thesis the popular CDN *jsDelivr* will be used, providing the benefits of a convenient API, i.e. a ranking of the most requested packages, providing an entry point for NPM packages and only serving code relevant to client-side execution. This eliminates the burden of creating a classification system to determine if source code is relevant for the browser environment, if it is popular or used at all as well if it is functionally intact or possibly broken and abandoned.

Based on *Assumption 3*, *Assumption 5* and *Assumption 6* all relevant versions for a package should be added to the corpus, i.e. excluding versions that do no not follow semantic versioning conventions and are deemed suitable for production. According to *Assumption 7,* this is necessary as old versions tend to be still in use, even when newer versions are available, and due to *Assumption 5*, their representation remains important in the corpus to not misidentify the associated license or security concerns. By excluding versions based on *Assumption 3* and *Assumption 6* the amount of data to be downloaded and stored is drastically minimized, as many packages come with daily or nightly builds published as new versions.

Based on *Assumption 9* it is only necessary to download the file provided by the CDN's entry point API, drastically reducing the amount of data to download, store and analyse as packages often contain large amounts of directories and files, e.g. version 3.3.1 of the library *date-fns* contains 4325 files.

Based on *Assumption 8* both the minified and non-minified variants of a library should be added to the corpus if available to make the corpus as representative as possible. The CDN *jsDelivr* provides a minified file by default, if the file identified at the entry point is not already minified it will be minified on the fly using either *Terser* (Santos & Vicente, 2018) or *UglifyJS* (Bazon, 2012). Having both minified and non-minified variants included in the corpus also allows for analysis of source code components that are so characteristic they exist in both variants, e.g. some Strings.

Based on both *Assumption 8* and *Assumption 4*, using a tool for clone search robust to modifications, such as stated in *Assumption 8*, for identifying an unknown library and then applying another technique to distinguish between the versions of said library seems sensible.

5.2 Implementation of the Corpus Builder

To construct a corpus, that unknown libraries on a given website can be compared against, the corpus builder application downloads JavaScript code for client-side execution. It downloads all versions that are hosted on NPM and considered suitable for production for the 1500 top-ranked downloads of the CDN *jsDelivr* for the period of the year 2023, this results in a corpus spanning 1378 libraries with multiple versions each.

5.2.1 Download

As briefly described in Section 4.3 the package download consists of the classes *Downloader*, *DownloadTrio* and *Reader*.

Instances of the class *Downloader* are created as download tasks for parallel execution. Its *call* method checks if the parent directory of the file path exists and if not, it creates all necessary directories in the path. Then it checks if the file already exists and if found returns it directly to avoid downloading the same file twice. This also helps avoid inconsistencies between the corpus and the index created by *Siamese* in case Assumption 3 does not hold and different variations of a source code file are published under the same package name and version number. If the file does not exist yet, the file content is downloaded via *ReadableByteChannel* from the URL and then transferred into the file via *FileOutputStream*. The file containing the downloaded content is then returned.

The class *Reader* consists of two similar static methods, offering the functionality to read from the file specified by the argument file path. The method *readFileToString* reads the content of the file into a String by utilizing the *java.nio.file.Files.readAllBytes* function and then casting the bytes to String. If an error occurs an error message and the stack trace will be printed. The read content is returned. The method *readFileToSet* is identical except for using *java.nio.file.Files.readAllLines* instead and adding the resulting List to a Set which is then returned.

The class *DownloadTrio* defines a data structure for download tasks. The constructor sets the class attributes *filename* specifying where the downloaded file should be stored, the URI of the resource to be downloaded and the Future<File> download, used to track the completion of the download task. The method blocking *hasTerminated* waits until the computation of the Future object representing a file download has finished and returns *true* if the download was successful, *false* otherwise.

5.2.2 Transform

The package *transform* consists of the classes *LibJSON* and *SaveToDisk*, as mentioned in Section 4.3

The class *LibJSON* relies on the *javax.json* package to process and manipulate JSON objects, offering five public static methods and the private helper function *readFromDir* which is used by the other methods to retrieve the paths to all the *.json files in a given directory.
The method *extractNames* takes no arguments but relies on the target directory and file naming for the top x pages specified in the config, iterating through the specified amount of pages and analyzing them for library names leveraging *javax.json.JsonPointer*. All detected library names are appended to a List of Strings and returned.
The method *extractVersions* works similarly to *extractNames*, also leveraging *javax.json.JsonPointer* to detect available versions, adding them to a List of Strings. It does require some further processing of the inspected *.json files as their structure is more complicated and it is necessary to iterate through JSON Arrays. Before the List of Strings containing the detected versions is returned, they are passed to the method *filterRelevantVersions*, which compiles and applies a Regular Expression pattern to filter out versions not suitable for production, e.g. containing tags such as canary, release candidate etc. If a library has no versions that are deemed suitable by the filter, the fallback option is to ignore the filter in this case.

The method *extractEntrypoints* checks the entry point files downloaded for each version of every library if they contain an entry point for JavaScript or Typescript and returns the retrieved entry points as a List of String Arrays. Some files contain only entry points for CSS, they will be ignored as they are not relevant to the corpus.
The method *extractNonMinifiedFromMap* takes a source map file and a target filename as arguments. The source map file is in JSON format, and the non-minified source code – which is saved as the value to the key "sourcesContent" – is extracted and saved to the target file using *SaveToDisk.saveCodeToFile*.

The class *SaveToDisk* has three static methods for saving data to a specified file: The method *saveJsonToFile* takes the target filename and a JSON object as input. First, it checks whether the target file already exists, if not, all necessary non-existing directories in its path are created. Then, using the *java.nio.file.Files* package, the data is written into the target file. If successful, the boolean *true* is returned, otherwise *false* and the stack trace is printed.
The methods *saveJobsToFile* and *saveCodeToFile* are very similar to the previously described method, except their second argument comes as a different data type and requires transformation before being written into the file: The method *saveJobsToFile* receives a list of string arrays as data instead of a JSON object, which is transformed utilizing a map function on a stream to first join the string arrays with " -> " as a delimiter and then joining all the resulting strings with '\n' as a delimiter. For the method *saveCodeToFile* the data comes in a List of Strings which are joined on a '\n' delimiter before being written to the file.

5.2.3 Execute

The class *config* is a config file realized as a *.java file instead of a properties, XML, or YAML file. While this would not be a good choice for a production system, where rebuilding the application to change the configuration would be unfavourable towards maintainability, it was a sensible choice for the prototypical implementation made for this thesis. The benefits include enabling programmatic generation or setting of values in the configurations as well as using compile-time checking to ensure correct typing.

The class *App* is the core of the application, leveraging the classes and their functions of the packages *download* and *transform* to build the corpus. Other than the *main* method, the class consists of further nine methods

The method *download_library_list* downloads a specified number of files in parallel, each containing the top 100 names of the libraries to be downloaded for the corpus. First, an instance of *java.util.concurrent.Executors.newFixedThreadPool* for the number of threads as defined in *config* is created. Followed by iterating over the number of pages to be downloaded, as defined in *config*, each time calling the helper-method *download_top_lib_page* to submit a download task to the thread pool and obtaining a Future<File> object for each iteration. After submitting the tasks, the thread pool is shut down and termination is awaited.

The method *download_version_info* operates in a very similar manner, creating a thread pool and calling its helper method *download_lib_versions* to submit download tasks to it, shutting down the thread pool and awaiting its termination. The difference is introduced by the input argument: The method receives the library names as a List of Strings, then iterates over the list to submit a task for each library name and manipulates the name to replace path separator characters.

Similarly, the method *retrieve_entrypoints* takes a JSON object, containing the library names as keys and their respective versions as values, as argument and creates a thread pool, submitting download tasks for each version of every library contained in the JSON object. To deter-

mine the URI of the file to download, the method iterates in two nested loops through all entries and constructs them by adding the library name and version to the String template defined in *config* for every iteration.

The method *download_all_code* is a simple wrapper for calling *download_minified_code*, *download_nonminified_code* and *download_mapfile_code*, storing their respective outputs in variables and printing some output, letting the user know about the progress of the downloads.

The methods are relatively similar in structure, they each take a list of string arrays as input, containing the downloads to be performed. They all create a thread pool, a list containing objects of type *DownloadTrio*, and two Lists of String Arrays; one for the failed and one for the succeeded download jobs. They iterate through the list provided as input, submitting a download task in each iteration, constructing an instance of the *DownloadTrio* type with the download task and the obtained Future<File> object, and then adding it to the list of downloads to check afterwards. After the thread pool is shut down and has terminated, the state of each download task is checked by iterating through the List<*DownloadTrio*> calling the *hasTerminated* method on every item, saving it to the list of succeeded tasks if *true* is returned or to the list of failed tasks otherwise. Both lists are then saved, to a location specified in *config*, by calling *SaveToDisk.saveJobsToFile* for debugging and auditing purposes.

The method *download_minified_code* follows exactly the previously described procedure, finally returning the list of succeeded jobs. As per the CDN *jsDelivr* API, the entry point always indicates a minified library, so no guarantees on the existence of a non-minified version are given. Therefore the method *download_nonminified_code* is called with the list of succeeded download tasks that was returned by the method *download_minified_code*. Since some of the minified source code files are generated by the CDN, the heuristic of removing the string ".min" from the URI to retrieve the non-minified file has been somewhat successful. This string manipulation is performed before passing the download task to the thread pool. Other than *download_minified_code*, this method returns the list of failed tasks.

The failed download tasks of non-minified source code are then used as input for the method *download_mapfile_code*, to try an alternative approach of retrieving the non-minified code: If the minified version of the file was not generated by the CDN but by the developer, often build tools were used, generating a source map file with the file ending ".map". So if the non-minified file was not present at the speculated URI, the method tries to retrieve the source map from yet another speculative URI. If successful, the non-minified source code can then be extracted from the file using *LibJSON.extractNonMinifiedFromMap*. Due to the non-uniform structure of code repositories, guessing the URI by some promising heuristic is not to be avoided without downloading the complete repository. The latter still offers no assurance of retrieving the non-minified source code, as some developers simply choose not to publish it.

5.3 Leveraging Siamese for Code Clone Identification

To identify possible code clones the tool *Siamese* is employed: *Siamese* receives source code as well as a config file as input for indexing and queries. It then returns the results of a query in the specified format, either CSV or GCF. The difficulties and the workarounds to configure and use *Siamese* are discussed in the following sections.

5.3.1 Configuration

While the general setup for the use of *Siamese* is well documented, the config file needed to be adjusted to be used with JavaScript. The config file is attached in Appendix A.

Specifically, the option *recreateIndexIfExists* caused some confusion: If set to *false*, no index will be created for the indexing task. If set to *true*, an existing index will be overwritten; so for large datasets requiring multiple indexing tasks, the option needs to be set to *true* for the first task and then changed to *false* for the following tasks. Alternatively, an index could be created manually, and the option should be set to *false*.

The option *minCloneSize* specifies the minimal clone size in lines. For a similar task as performed by Misu and Satter (2022) a value of *10* was chosen, but for this value all minified files consisting of only one line of code would be ignored. To account for minified files, often only containing a single line of code, this value has to be set to *one* for queries.

5.3.2 Other Problems

Siamese, like many other Java applications, is very greedy when it comes to heap utilization. With a steadily rising memory allocation and the crash of the application with the *java.lang.OutOfMemoryError* exception being thrown, this could indicate a memory leak. To circumvent this issue, the dataset for indexing has been split into smaller subsets and corresponding indexing tasks.

Siamese relies on an ANTLR Grammar – generating a corresponding lexer and parser – to perform lexical analysis, code that is not compatible with the grammar will cause Syntax Errors to be thrown and the incompatible code will not be added to the index with multiple representations.

The source code of *Siamese* contains a hard-coded relative path to the data folder of *ElasticSearch*. If the data and logs of *ElasticSearch* are configured to be stored in another location than the application itself, as I configured it on my setup, this path in *src/main/java/crest/siamese/Siamese.java* line 1063 needs to be modified.

5.4 Benchmark

As already identified in Chapter 2 , there is a lack of benchmarks for code clone identification in JavaScript source code. The considerations for the creation of a small benchmark will be laid out in the following sections.

5.4.1 General Considerations for Creating a Benchmark

Benchmarks can increase transparency as they promote a collaborative, open, and public conduct of research. The creation of benchmarks can help advance the maturity of a scientific community. (Sim, Easterbrook, & Holt, 2003)

Sim et al. (2003) identify seven properties of successful benchmarks:

- **Accessibility**: The benchmark needs to be easy to obtain and use.
- **Affordability**: The benefits of the benchmark must outweigh the cost of using the benchmark; hardware, human resources or other.
- **Clarity**: The specification of the benchmark must be clear, self-contained and as short as possible.
- **Relevance**: The task defined by the benchmark must be representative of a reasonably expected problem.

- **Solvability**: The task should be achievable but non-trivial to gather enough data for comparison.
- **Portability**: The benchmark needs to be portable to different tools or techniques and should not bias one technology in favour of another.
- **Scalability**: The tasks should scale to work with tools or techniques with different levels of maturity.

Existing benchmarks for JavaScript, e.g. the no longer maintained) *SunSpider* benchmark (WebKit) or the more comprehensive *JetStream* benchmark (Apple), tend to focus on measuring the capabilities or performance of various JavaScript engines.

Richards et al. (2011) created the tool *JSBench* to automatically create such benchmarks for comparing and tuning the performance of JavaScript interpreters and just-in-time compilers. This tool manages to capture a program, including its dynamically generated components, and replay it deterministically, allowing the creation of a benchmark from real-world applications. Richards et al. argue that a framework for the automatic creation of a benchmark is especially valuable for a rapidly evolving technology such as JavaScript web applications.

Derks, Strüber, and Berger (2023) created the framework *vpbench* to generate benchmarks for so-called variant-rich software "by automatically adding, removing, and cloning features, mutating implementation assets (e.g., code), and cloning variants" (Derks et al., 2023, p. 2). This framework is created for Java but claims to be language independent and extendable for another language by applying changes to the parser, providing a tool to check for successful compilation, the feature transplantation process as well as the process of cloning features. While this sounds like a promising approach, it would take a lot of time and effort to make *vpbench* work for clone detection in JavaScript, making it not reasonably fit into the scope of this thesis.

5.4.2 Creating a JavaScript Benchmark for Code Clone Identification

The most controversial decision in creating any benchmark is choosing which data and tasks to include. The goal of this benchmark is to accurately represent client-side JavaScript code, therefore the following criteria, as already superficially discussed in Chapter 3 should be considered:

- Multiple ECMAScript versions should be represented with language constructs specific to their version. This ensures that the clone search tool can deal with realistic input data.
- The benchmark should contain source code transpiled to ES5, as is typically done to ensure browser compatibility.
- Modifications should be applied to source code before transformations are applied, to emulate copy, clone and edit behaviour of developers.
- The transformations on the source code should be performed by popular tools that would be realistically used for websites:
    - For minification, a popular choice of tool would be *terser* (Santos & Vicente, 2018),
    - For obfuscation the tool *javascript-obfuscator* (Kachalov, 2016) could be used,
    - For bundling the tool *webpack* (Koppers et al., 2016) is extremely popular,
    - For transpilation the tool of choice is *Babel* (Nicolini et al., 2024).

Due to the time constraints of this thesis, it was not possible to develop a full-fledged benchmark. The developed proto-benchmark contains the data as listed in Table 1. Notably, no bundles have been included due to the effort required to create a set of bundles representative of the settings and optimization mechanisms offered by *webpack* as well as covering multiple combinations of source files being collated by the tool, while auditing how much of the libraries is really included in the resulting bundle, due to mechanisms such as tree shaking removing code that is not required.

The reasoning behind choosing these libraries is as follows:

The library *jquery* is included as it is one of the most popular libraries. Two versions, the earliest and the latest version included in the corpus, are included in the set to see if the tool is capable of discriminating between the two versions. *Vuetify* is included as it contains a variety of ECMAScript language features, as shown in Appendix C. *Typescript* is a highly popular library which was included for its extraordinary length: Version 5.0.2 consists of 169788 lines of code - including comments and whitespace – which makes it interesting to observe how it will be affected by the transformations in contrast to the very small library *whatwg-fetch*, which only spans 230 lines of code in version 0.5.0.

The results of the queries performed on this benchmark are presented and discussed in Section 6.1

Library	Version	Variant	Tool	Settings
jquery	1.5.1	original	N/A	N/A
		original minified	unknown	unknown
		minified	terser v5.16.5	--keep-fnames --compress --mangle
		obfuscated	javascript-obfuscator v4.0.0	preset = LOW
		modified	manually	introducing syntax error
	3.7.1	original	N/A	N/A
		original minified	unknown	unknown
		minified	terser v5.16.5	-- compress --mangle --rename
		obfuscated	javascript-obfuscator v4.0.0	preset = MEDIUM
vuetify	3.0.0	original	N/A	N/A
		original minified	unknown	unknown
		minified	terser v5.16.5	--comments --compress --mangle
		obfuscated	javascript-obfuscator v4.0.0	preset = MEDIUM
		transpiled to ~ES5	babel v7.20.7 (@babel/core v7.20.12)	@babel/preset-env: targets: browsers: ie<8
whatwg-fetch	0.5.0	original	N/A	N/A
		original minified	UglifyJS v3.1.10	unknown
		minified	terser v5.16.5	--compress --mangle
		modified + minified	terser v5.16.5	--compress --mangle
		obfuscated	javascript-obfuscator v4.0.0	preset = HIGH
		modified	manually	renaming variables, whitespace changes
typescript	5.0.2	original	N/A	N/A
		original minified	Terser v5.15.1	unknown
		minified	terser v5.16.5	--keep-fnames --comments --compress --mangle

Table 1: JavaScript libraries and transformations performed for the benchmark

6 Evaluation

In this chapter, the results obtained on the benchmark, as defined in Section 5.4.2 will be presented and discussed. Then, the corpus builder, and suitability of the clone search tool *Siamese* and corpus will be evaluated against the requirements, functional and non-functional, as defined in Chapter 3 Every requirement will be evaluated as either fulfilled, partially fulfilled or not fulfilled.

6.1 Results on the Benchmark

To process all the source code files in the benchmark, as described in Section 5.4.2 or Table 1, they are first indexed by *Siamese* and then each queried. Using the exact same set of source code files for indexing and querying provides a sanity check, even if a file can not be matched with other variants of the same library, it should always be matched to its exact copy in the index, provided *Siamese* was able to process the source code file correctly. The indexing phase was performed twice, once in method mode and once in file mode, storing data in two different clusters. The indexing in method mode resulted in 7968 documents or 71.6 MB of data stored in the *ElasticSearch* index, the file mode resulted in 23 documents or 28.1 MB of data stored in the index. For each library in the benchmark, an individual query was performed, generating a CSV file containing multiple lines, each line containing up to eight clone candidates for a fragment of the queried code. The choice of configuring the threshold of top results for queries to return is a result of the small size of the benchmark. A number too large relative to the benchmark size of 23 files could result in all files being returned as a query result, diminishing the information gained by the query. A number much smaller than eight would remove the possibility of investigating how well Siamese can discriminate between different versions of *jquery* or between different variants of *whatwg-fetch* while still intentionally allowing some confusion with other libraries to be detected. This threshold should be set considering the size of the index the queries are performed on.

The *ElasticSearch* parameter *indices.query.bool.max_clause_count* has been set to 32000 instead of the default value of 4096.

The query on the index created in file mode did not return any results, all generated output CSV files were empty. Error output was only generated for some of the obfuscated files, i.e. the obfuscated variants of *jquery@1.5.1*, *jquery@3.7.1* and *whatwg-fetch@0.5.0*, indicating that the serialization of the query failed due to too many clauses or the parameter *indices.query.bool.max_clause_count* being set too low.

The result of the query on the index created in method mode is represented as a matrix in Table 2, showing every permutation of library variants if *Siamese* identified them as potential code clones. All of the queries were also performed on the index created in file mode, but none of the queries returned results.

The results in Table 2 are to be interpreted as follows:

- Yellow horizontal: The query for this library did not return any results and an error output was printed.
- Dark grey horizontally striped: The query for this library did not return any results but no error output was printed.
- Dark green fields: The query for this library returned the correct library and the identical variant.
- Light green fields: The query for this library returned another variant of the correct library.

- Red fields, diagonally striped: The query identified other libraries as potential code clones.
- Numbers: The number equals the number of occurrences of a library variant in the CSV file containing the results. The numbers are not normalized, so they should only be compared intra-row-wise.

		Returned Library	jquery									whatwg-fetch						typescript			vuetify				
		Version	1.5.1					3.7.1				0.5.0						5.0.2			3.0.0				
Query Library	Version	Variant	original	original minified	minified	obfuscated	modified	original	original minified	minified	obfuscated	original	original minified	minified	modified + minified	obfuscated	modified	original	original minified	minified	original	original minified	minified	obfuscated	transpiled to ~ES5
jquery	1.5.1	original																							
		original minified		1370	199			8	37	20															
		minified		217	1773			9	192	210															
		obfuscated																							
		modified																							
	3.7.1	original		26	26	3		1690	26	26	1						2								
		original minified		48	270			24	458	973															
		minified		40	334			24	930	448			1	2	1										
		obfuscated																							
whatwg-fetch	0.5.0	original						3				103	5	6	2		9								
		original minified							6	8		6	33	68	63		1								
		minified							6	8		6	61	33	77		1								
		modified + minified							8	10		2	56	77	31		1								
		obfuscated																							
		modified						5				10					95								
typescript	5.0.2	original																							
		original minified																							
		minified																							
vuetify	3.0.0	original																							
		original minified																							
		minified																							
		obfuscated																							
		transpiled to ~ES5																							

Table 2: Siamese's top 8 query result matrix

From Table 2 it can be observed that no queries for any variants of the libraries *vuetify* or *typescript* returned any results, nor were they returned in any result of the other queries performed. As no error output was generated for these files, neither in the indexing nor in the query phase, without in-depth debugging of *Siamese* it can only be speculated where and why the quiet failure occurred.

This behaviour can also be observed for some variants of other libraries, including the original and the modified variants of *jquery@1.5.1* as well as the obfuscated variant of *jquery@3.7.1*, except the latter was once included in the result of the query for the original variant of *jquery@3.7.1*.

		***Siamese* Setting**	
		query reduction	**no query reduction**
***ElasticSearch* Setting**	max_clause_count = **32k**	Error	Error
	max_clause_count = **128k**	No Error / No Results	No Error / No Results

Table 3: Output for the query of obfuscated jquery@1.5.1 depending on Siamese and ElasticSearch settings

For the obfuscated variants of *jquery@1.5.1* and *whatwg-fetch@0.5.0* error output was generated by *ElasticSearch*, indicating a *SearchParseException*, i.e. that the serialization of the query failed due to too many clauses or the parameter *indices.query.bool.max_clause_count* being set too low, as observed for the index generated in file mode. A plausible origin for the failure with error output happening during the query phase but not the indexing phase, as marked by the yellow horizontal lines in Table 2, could be the query reduction, since the processing of input data is identical during the indexing phase and query phase other than the query reduction step. Looking into the generated *SearchParseException* revealed that the cause is often a malformed query, raising the suspicion that *Siamese*'s query reduction module is not able to handle obfuscated source code. To investigate this, queries on the obfuscated variant of *jquery@1.5.1* have been performed with the parameter *indices.query.bool.max_clause_count* first set to 32k and then 128k and both options with and without query reduction enabled, see Table 3. While the use of the query reduction module seems to have no impact on the outcome, the increase of the parameter *indices.query.bool.max_clause_count* to 128k resolved the error, but still, no results were returned for the query. This has been performed on both the index generated in file mode and the index generated in method mode, leading to the same results. Why the query did not at least match itself as a clone pair, as one would anticipate, remains unclear.

The queries for the libraries *jquery* and *whatwg-fetch* produced some more conclusive results, between *jquery@3.7.1* and *whatwg-fetch@0.5.0* some similarities were detected by *Siamese*. To investigate these potential similarities, the original variants of both libraries were investigated utilizing *Moss* (Aiken): Comparing them without specifying a language, a 3% similarity was detected due to a code fragment where values were set for differently named attributes in both source code files. The comparison with the language set to JavaScript did not detect any significant similarities. To discuss the intra-library query results for both libraries Table 4 and Table 5 only show the relevant subset of Table 2.

In Table 4 *Siamese*'s ability to discriminate between different variants of the library *jquery* is displayed. The modified variant of version 1.5.1 contains syntax errors, no results for its query were returned nor was it included as a possible clone pair in any other queries – possibly because it could not be processed correctly by *Siamese* due to the syntax errors.

The behaviour for the original variant of version 1.5.1 is identical, but no rationale comes to mind that would explain why the query did not pass the sanity check of at least identifying the variant itself as a match when other variants of this library were able to be processed correctly by *Siamese* and produced conclusive results. The original variant of version 3.7.1 is matched with all variants except the original and modified variants of version 1.5.1, but most strongly with itself. While this clearly identified the library, the minified variants of both versions are equally represented, indicating *Siamese* might not be able to discriminate between two versions of the same library.

		Returned Library	jquery								
		Version	1.5.1					3.7.1			
Query Library	Version	Variant	original	original minified	minified	obfuscated	modified	original	original minified	minified	obfuscated
jquery	1.5.1	original									
		original minified		1370	199			8	37	20	
		minified		217	1773			9	192	210	
		obfuscated									
		modified									
	3.7.1	original		26	26	3		1690	26	26	1
		original minified		48	270			24	458	973	
		minified		40	334			24	930	448	
		obfuscated									

Table 4: Siamese's top 8 query results for jquery

		Returned Library	whatwg-fetch					
		Version	0.5.0					
Query Library	Version	Variant	original	original minified	minified	modified + minified	obfuscated	modified
whatwg-fetch	0.5.0	original	103	5	6	2		9
		original minified	6	33	68	63		1
		minified	6	61	33	77		1
		modified + minified	2	56	77	31		1
		obfuscated						
		modified	10					95

Table 5: Siamese's top 8 query results for whatwg-fetch

The queries for all minified variants of both versions each return all minified variants of both versions and the original variant of version 3.7.1 as clone pairs. The query on minified variants of version 1.5.1 scored themselves the highest, and the variants of version 3.7.1 each confuse themselves with the other minified variant – possibly due to similar minification settings. As already mentioned, the obfuscated variants of both versions did not return any query results, but both were included in the query results of the original variant of version 3.7.1.

Analogue to Table 4, in Table 5 *Siamese*'s ability to discriminate between different variants of the library *whatwg-fetch* is exhibited. The query on the original variant clearly identified the original variant as a clone pair, as anticipated, and also matched with the other variants except the heavily obfuscated variant. All three queries for minified variants, original minified, minified, and modified and then minified, each identified themselves as a possible clone pair but were confused with the other minified variants. They also recognized some similarities with the original and the modified variant. The query for the modified variant identified itself as a clone pair and detected some similarity with the original variant, but no minified variants were identified as potential clone pairs, creating an asymmetrical result matrix. No results but error output was returned for the query on the obfuscated library, as previously discussed.

6.2 Evaluation of Functional Requirements

The functional requirements to be evaluated have been defined in Section 3.1

Corpus:

F-01 The requirement is partially fulfilled, the corpus does contain source code such as CommonJS modules that cannot be natively executed by browsers but is still relevant for client-side execution.

F-02 The requirement is fulfilled, and the corpus is somewhat representative. The goal of building a representative corpus is already built into the creation process, as the 2023 top 1500 packages requested from the CDN *jsDelivr* are inspected and downloaded. To assert that this is indeed a representative sample, a comparison against the Wappalyzer top 100 JavaScript libraries ranking for 2023 (Wappalyzer Pty Ltd, 2023b) is conducted, see Appendix B.

According to this comparison, 60 of the 100 libraries listed by Wappalyzer are also included in the corpus, covering 84% of the market share of the top libraries identified by Wappalyzer. Some of the libraries that were missing in the corpus are not hosted on NPM, e.g. Microsoft Authenticator. The rankings of CDN *jsDelivr* and Wappalyzer also differ because of different methodologies used when constructing the rankings: Wappalyzer ranks by the number of websites the library was detected on using their browser extension, while *jsDelivr* ranks by the number of requests they receive for a library. According to this comparison, the constructed corpus seems to be representative of the 2023 webscape.

F-03 The requirement is fulfilled, the corpus contains source code covering multiple ECMAScript versions. To assert that various ECMAScript versions are represented by the corpus, a sample of newly introduced language features for recent ECMAScript versions has been selected to indicate the presence of the corresponding version. The corpus has been analyzed for said features, some examples detected in the corpus are listed in Appendix C.

F-04 The requirement is partially fulfilled, the corpus contains all versions of a library available on NPM, nevertheless, versions are removed from NPM from time to time.

Corpus Builder:

F-05 The requirement is fulfilled, the corpus builder does allow for easy extension of the corpus by making small adjustments to *config* and downloading only library versions that are not yet locally stored.

Tool for Clone Search:

F-06 The requirement is not fulfilled, *Siamese* is seemingly able to process and index a representative corpus of JavaScript client-side libraries, not creating any output indicating otherwise. The tool claimed to have successfully indexed the corpus as well as the benchmark sample, creating all four representations only for a subset of compatible files. The query phase was not as tolerant towards non-compatible files, resulting in no code clones identified or error messages, revealing that the file representation was useless for any queries performed on the benchmark.

F-07 The requirement is partially fulfilled, *Siamese* can produce somewhat human-readable output: It offers GCF or CSV as output formats, unfortunately, GCF does not work for JavaScript and the CSV is not as comprehensible without further postprocessing.

F-08 The requirement is partially fulfilled, *Siamese* takes a config file as input, but unfortunately the flexibility in configuring *ElasticSearch* is very limited without editing hard-coded paths in the *Siamese* source code.

F-09 The requirement is partially fulfilled, as *Siamese* is able to detect code clones on minified source code, provided it was able to successfully process the source code.

F-10 The requirement is not fulfilled, *Siamese* is not able to detect code clones robustly on obfuscated source code. Queries on obfuscated source code resulted either in error output or no query results being returned. Queries on non-obfuscated source code rarely identified the corresponding obfuscated source code.

F-11 The requirement cannot be evaluated, due to ***F-19*** not being fulfilled.

F-12 The requirement cannot be evaluated, *Siamese* was not able to detect code clones of the transpiled source code file, as it was not able to process the source code file correctly.

F-13 The requirement is partially fulfilled, *Siamese* is able to detect code clones on modified source code, provided the source code file can be processed successfully by Siamese, i.e. the source code is not syntactically broken.

JavaScript Benchmark:

F-14 The requirement is fulfilled, the benchmark contains JavaScript client-side source code in various ECMAScript versions.

F-15 The requirement is fulfilled, the benchmark contains the library *vuetify@3.0.0* transpiled to an equivalent of ES5, see Table 1.

F-16 The requirement is fulfilled, the benchmark contains manually modified JavaScript client-side source code; one modified variation intentionally introducing syntax errors and one syntactically sound modified variation, see Table 1.

F-17 The requirement is fulfilled, the benchmark contains minified JavaScript client-side source code. The benchmark contains both the minified source code distributed by the CDN as well as source code minified by *terser* (version 5.16.5) (Santos & Vicente, 2018) with different minification settings, see Table 1.

F-18 The requirement is fulfilled, the benchmark contains obfuscated JavaScript client-side source code. The libraries *jquery* (versions 1.5.1 and 3.7.1), *vuetify* (version 3.0.0) and *whatwg-fetch* (version 0.5.0) were obfuscated by the tool *javascript-obfuscator* (version 4.0.0) (Kachalov, 2016) for the benchmark.

F-19 The requirement is not fulfilled, the benchmark does not contain any bundled JavaScript client-side source code.

6.3 Evaluation of Non-Functional Requirements

The non-functional requirements to be evaluated have been defined in Section 3.1

Corpus Builder:

NF-01 The requirement is fulfilled; the corpus builder application is written in Java.

NF-02 The requirement is fulfilled, the corpus builder application efficiently downloads packages in parallel and avoids unnecessary network traffic by avoiding duplicate downloads, downloading only what is not yet stored on disk.

Tool for Clone Search:

NF-03 The requirement is fulfilled, *Siamese* is licensed under the GNU General Public License (GPL-3.0) and the Apache 2.0 license.

7 Conclusions

In this chapter, the conclusions drawn from the literature review, as presented in Section 4.1 will be recapped and the insights gained from experimentation with *Siamese* will be examined. Possible threats to the validity of the experiments performed and conclusions drawn will be discussed in Section 7.2 and finally, potential research directions and future work on this topic will be outlined in Section 7.3 .

7.1 Conclusions

To recapitulate the findings of the literature research performed in Chapter 2 the identification of JavaScript libraries and their version remains an active field of research. The domain of the browser as well as the dynamic nature of JavaScript, paired with the continuously evolving ECMAScript specification make it a difficult task and cause developed tools to become obsolete quickly. No popular benchmarks or frameworks for benchmark generation for code clone detection in JavaScript source code have yet emerged.

Several techniques have been further investigated in section 4.1 and *Siamese* (Ragkhitwetsagul & Krinke, 2019) has been selected as the tool of choice for this thesis. A corpus builder was implemented and used to construct a corpus containing all versions suitable for production and hosted on NPM for the most popular JavaScript libraries via the CDN *jsDelivr*. *Siamese* was then used to index the corpus, creating a searchable *ElasticSearch* index containing multiple representations of each library for clone detection. To evaluate Siamese's suitability for the identification of client-side JavaScript libraries, a small proto-benchmark has been constructed in Section 5.4.2 and the results have been analysed in Section 6.1 .

While *Siamese* was the best choice of tools presented in Chapter 2 and has already been successfully used to conduct a study on JavaScript code clones, the evaluation of the achieved results and the requirements, as defined in Chapter 3 and evaluated in Chapter 6 revealed that *Siamese* is not an ideal choice for the identification of client-side JavaScript libraries.

Siamese did prove not to be very user-friendly, especially for indexing a large corpus; It required modifications to the source code to change hard-coded paths, and the application was too memory intensive, requiring the indexing to be split into multiple smaller tasks to avoid crashes caused by memory leaks. During the indexing phase some output was generated, indicating that *Siamese* was struggling to process the syntax of modern libraries correctly, therefore defaulting to not creating any alternative representation for those libraries and just adding them as a file to the index.

When trying to parse the code in the corpus the ANTLR grammar used by *Siamese* was not sufficient, updating the grammar as well as ANTLR itself to a later version did not resolve the issue, leading to the observation that JavaScript might not be representable by a context-free grammar anymore. The current ECMAScript specification confirms the lexical grammar now contains some context-sensitivity (ECMA TC39 Committee, 2024).

Experiments with the proto-benchmark in Section 6.1 revealed that none of the libraries indexed in file mode could be successfully used to identify any potential code clones, making part of the index generated on the corpus unusable. While *Siamese* generates some output, the resulting behaviour is still far from explainable and the tool could be more verbose. The query results are available in the formats CSV and GCF, although GCF is not supported for JavaScript and the two variations of CSV formats could be improved in regards to interpretability and legibility for humans.

The more successful queries, executed in Section 6.1 were able to pair minified and non-minifed source code together but were not able to discriminate between different versions of one library. *Siamese* was not able to reasonably process any obfuscated source code or syntactically broken source code included in the benchmark, emphasizing the need for more verbose output and better explainability of results.

Considering the fast evolution of JavaScript, writing a new parser and lexer from scratch and then maintaining them for new ECMAScript versions is not a sensible solution. It would be more practical to rely on a browser engine to generate the AST, e.g., *V8* or *SpiderMonkey*, as they will continue being maintained to work for the latest ECMAScript specification. According to the ECMAScript compatibility table (Zaytsev, Pushkarev, & et al, 2023) both Mozilla Firefox, using *SpiderMonkey*, and Google Chrome, using *V8*, cover 98% of ECMAScript features at the time of writing.

Unfortunately, even with a modern, more flexible AST-based approach, *Siamese* would still be inherently susceptible to misclassification due to obfuscation techniques that change the structure of the AST, such as control flow flattening and dead code injection. Even if just a small code fragment is obfuscated, it could contain inclusions that evade detection from static analysis, rendering the analysis or SBOM of the whole website non-declarative. While static analysis approaches can be outmaneuvered by obfuscation, researchers concerned with internet security and privacy have come up with dynamic analysis techniques to monitor the behaviour of the obfuscated code, i.e. Ngan, Konkimalla, and Shafiq (2022) and Le, Fallace, and Barlet-Ros (2017) investigated techniques to identify webtracking or fingerprinting hidden by obfuscation.

To summarize, *Siamese* can identify client-side JavaScript libraries, but only to a very limited extent and quiet failures of the tool undermine its explainability as well as credibility. It is necessary to investigate alternative approaches that are able to accommodate modern ECMAScript language features as well as code transformed by minification and obfuscation tools.

7.2 Threats to Validity

To ensure factual accuracy, the assumptions based on personal observations made due to the lack of reliable sources have been explicitly stated in Section 5.1.1 Any other assumptions that were only implied and could threaten the integrity of the thesis are outlined below:

This thesis builds on the assumption of the correctness of information sources, specifically that the CDN *jsDelivr* contains the version of a library that it claims.

This tool constructed for this thesis only adds library versions to the corpus that are still available on NPM due to the use of the *jsDelivr* API only listing those versions. From the gap between version numbers and the previous experimentations with different APIs (e.g. the *CDNJS* API (Kirkman, Davis, Cowley, Sauleau, & Caslin, 2011)) it has become clear that deletion of specific library versions is not uncommon. Since deleting a version on NPM does not mean this version will not be used by websites already in production, this might create a blind spot in the corpus with regard to certain versions.

The statistics provided by the CDN *jsDelivr* measuring the popularity of JavaScript remote inclusions might not be representative of their actual popularity, as they can only provide information on the usage of inclusions via the CDN *jsDelivr*, not other CDNs or inclusions through local copies.

The statistics provided by Wappalyzer Pty Ltd (2023b) on the market share of JavaScript libraries on websites should be taken with a grain of salt: The methodology on how Wappalyzer gathered the data is not publicly available and can only be obtained by closer inspection of

their browser extension. The analysis of the browser extension showed they perform both static and dynamic analysis, as described in Section 4.1

The ranking of the top 15 pages of 2023 on *jsDelivr* as well as the corresponding libraries were downloaded on 08.01.2024, this means library versions that were not available yet in 2023 could be included in the corpus. Some of the packages listed in the aforementioned top 15 pages did not contain any endpoints for JavaScript, e.g. only a CSS endpoint, and were therefore dismissed.

When analyzing a small random sample of downloaded source code files, it became clear that some entry point files consisted only of one or more dynamic inclusions. To use these files for clone detection further steps are required to dynamically resolve the inclusions and obtain the code like it would be executed by the browser. This could be achieved by extending the corpus builder application.

Some libraries are nearly impossible to differentiate, as sometimes two different libraries share a common ancestor, e.g. *maplibre-gl-js* is a community-driven fork of *mapbox-gl-js*, which was previously licensed under a BSD-3-Clause license for versions 1.13 and earlier and is now no longer under an open-source license (Mapbox, Inc., 2023; “maplibre-gl-js LICENSE.txt,” 2023). While both libraries look very similar due to shared ancestry, their licenses are not, so a theoretical false identification of the library with high confidence – possibly caused by only one of the libraries being represented in the corpus – could lead to a completely wrong attribution with possible legal consequences.

The configuration of *Siamese* was not necessarily optimal as no experiments have been conducted to compare and optimize configuration settings for this thesis.

All statements made on JavaScript were correct to the best of my knowledge, based on recent literature. The data collected and statements made on JavaScript could already be outdated by the time of your reading, as JavaScript and JavaScript libraries evolve remarkably fast.

7.3 Future Work

Due to the time constraints of a master thesis and the vastness of the research field a lot of potential research questions remain.

The identification of the corresponding version for a detected library has not yet been implemented due to time constraints. Pagon et al. (2023) have introduced a static analysis technique of identifying the exact version of a library based on hashing with 50% success or the probable version range with a ~75% success rate. Pagon et al. claim their technique to be robust against minification, but not against obfuscation. Further research to improve the detection of the correct version number needs to be conducted as a correct classification in only 50% of cases still leaves plenty of room for improvement.

As mentioned in Chapter 2 as well as in Section 4.1 browser extensions can identify the version of a detected library even when minified using a knowledge base of extractors. Complementing the static approach presented in this thesis with a dynamic approach could be beneficial but requires further research into how the knowledge base containing the required extractors can be generated automatically.

Another crucial, yet still missing, step for the construction of an SBOM would be the extraction of the license information for the licenses that could be identified. *Siamese* already offers two options for the extraction of licenses: *Ninka* (German, Manabe, & Inoue, 2010) or using

regular expressions for extraction. Neither options nor the integration into *Siamese* has been tested in this thesis.

Finally, the generation of an SBOM or the interface to interact with existing tools offering the functionality to create an SBOM is yet to be realized, provided the necessary version information can be obtained.

Since the scope of this thesis was just covering the edge case of client-side code where neither the name of the file nor comments offered a reliable source for identification of a library and its version, the more straightforward case of simply comparing strings should be implemented. Also, the case of libraries not undergoing any transformations was disregarded, so adding a comparison of file hashes to identify unmodified files should be considered as well. Alternatively, the compression-based similarity could be investigated as a similarity measure to quickly identify Type-1 or Type-2 clones.

Lauinger et al. (2017) noticed that libraries sometimes differ between official websites and different CDNs: Additional whitespace, removal of comments and minification using different tools or settings can occur between the same version and variant of the same library. As previously stated, this thesis also builds on the assumption of the correctness of information sources, specifically that the CDN *jsDelivr* contains the version of a library that it claims. It would therefore be of interest to collect more data from other sources such as the library's website. To address the possible blind spot in the corpus with regard to certain versions, it would be of interest to supplement the corpus with more sources of data. If a version was available while the corpus was constructed, this is not an issue, but if it was already deleted off NPM additional data sources such as *npm-follower* (Pinckney, Cassano, Guha, & Bell, 2023) might be a valuable addition.

Also, compared to the ranking by Wappalyzer Pty Ltd (2023b) it became obvious that some popular libraries are not hosted on NPM at all, e.g. *Microsoft Authenticator*. While NPM seems to be a good source for third-party components under OSS licenses, proprietary software is usually not hosted there and would require screening for more relevant sources.

Due to the time constraints of the thesis, it was only possible to investigate code clones present in JavaScript code, disregarding the possibility of WebAssembly (Rossberg, 2019) as a compilation target for a multitude of programming languages. Compiler toolchains like *Emscripten* also make this possible by first creating an intermediate representation leveraging the Low-Level Virtual Machine compiler infrastructure and then translating it into WebAssembly code as well as JavaScript glue code (Zakai, Dawborn, Shawabkeh, & et al, 2015).

To identify code clones originating from another programming language, a cross-language clone detection approach is required. Further investigation on how the compilation to a different language affects the respective license agreements should be conducted.

To improve *Siamese*'s interpretability of modern JavaScript, specifically ES2015+, some modifications are necessary. The parsing and tokenization currently rely on ANTLR or rather on an ANTLR Grammar, which is not able to fully represent the language features introduced in later ECMAScript versions as they can no longer be represented by context-free grammar. More modern tools like a JavaScript engine used by browsers, e.g. *SpiderMonkey* (Mozilla Foundation), could be used to parse JavaScript code, generate an AST representation etc. This is not an uncommon approach, Bazon (2012) also seems to rely on *SpiderMonkey* to generate an AST for the popular minification tool *UglifyJS*.

Alternatively, a normalization by transpilation should be explored: It remains to be tested to use a transpiler such as *Babel* (McKenzie & et al, 2018) before passing the code to *Siamese*. The effects on the similarity relationship should be studied as the transpilation might skew the

similarity distance by introducing similar boilerplate code replacing previously different language constructs. Further normalizations to be considered include un-minifying and beautifying minified files before indexing them, similar to Pagon et al. (2023).

A much different approach that should be investigated is a bytecode-level approach, which has been successfully implemented for other programming languages, e.g. Java (Wan et al., 2023; D. Yu et al., 2019), but is yet to be applied to JavaScript. Since JavaScript is, unlike Java, not a language compiled ahead of time, but JIT compiled, obtaining the bytecode requires more effort and finesse. Bytecode has already been used to identify the insertion of malicious code (Rozi, Kim, & Ozawa, 2020) in JavaScript code, so analyzing the JavaScript bytecode to detect code clones seems feasible in principle, provided it is possible to overcome the burden to obtain the bytecode for a large number of libraries.

A different route for future research could be compiling JavaScript source code to WebAssembly, obtaining a textual representation as well as the Wasm binary format for clone detection. Experiments on modifications and normalizations on the source code before compiling to WebAssembly and how it affects the detectability of various clone types on the generated code should be performed. Using WebAssembly as a compilation target would offer the benefit of extendibility towards other programming languages as Type-4 clones could occur across languages.

Lastly, a benchmark suite to evaluate tools and techniques for clone search on JavaScript is urgently required. As discussed in Section 5.4.1 it would be very valuable to have a framework for the automatic generation of a JavaScript benchmark. Further investigation into *vpbench* (Derks et al., 2023) and the changes required to adapt to JavaScript and integrate tools for minification and obfuscation should be considered.

Although this thesis has addressed several aspects of identifying client-side JavaScript libraries, numerous avenues for further exploration and refinement remain. In addition to enhancing version detection techniques and extending cross-language clone detection approaches, this field also offers opportunities for bytecode-level analysis and leveraging WebAssembly as a compilation target. It would also be beneficial to develop a comprehensive benchmark suite to evaluate JavaScript clone detection tools and techniques. Considering the complexity and breadth of this research area, it is crucial to continue investigating and innovating to address the challenges and opportunities it presents.

Appendix A Siamese Config File

GENERAL CONFIGURATIONS

elasticsearchLoc=/var/tmp/ga79cace/elasticsearch # location of elasticsearch

server=localhost # elasticsearch's server name (or IP)

cluster=MA_AS-IS # elasticsearch's cluster name. See cluster.name in your

$elasticsearchLoc/config/elasticsearch.yml

index=idx # index name

type=siamese # type name

inputFolder=/var/tmp/ga79cace/280124_downloaded_code # location of the input

folder. This is the location of the files to be indexed, or the location of the queries

outputFolder=search_results/as-is-file-mode # output folder to store the search results

dfs=false # use DFS mode [default=no]

writeToFile=true

extension=js # source code file extension

minCloneSize=1 # minimum clone size (lines)

command=index # command to execute [index,search]

isPrint=true # print out logging data

outputFormat=csvfline # output format [csv = filename, csvfline = filename#start#end),

gcf = general clone format]

indexingMode=bulk # indexing mode [sequential, bulk]

bulkSize=100 # size of bulk insert

parseMode=file # clone granularity [method, file]

printEvery=250 # print the progress of indexing/querying in every x files

recreateIndexIfExists=false # recreate the index if it exists [true, false]

PARSER + TOKENIZER + NORMALIZER SETTINGS

methodParser=crest.siamese.language.javascript.JSMethodParser

tokenizer=crest.siamese.language.javascript.JSTokenizer

normalizer=crest.siamese.language.javascript.JSNormalizer

MULTI-REPRESENTATION SETTINGS

multirep=true

enableRep=true,true,true,true

NORMALIZATION MODE:

Code normalisation for T2 and T3 representation.

Combination of k (keywords), v (values), s (strings), o (operators), w (words)

normalizerMode=crest.siamese.language.javascript.JSNormalizerMode

t2NormMode=vsw

t3NormMode=kvsow

isNgram=true # turn on ngram

size of ngram.
ngramSize=4 # representation T3
t2NgramSize=4 # representation T2
t1NgramSize=4 # representation T1

QUERY-RELATED SETTINGS
resultOffset=0 # starting result offset (usually zero)
resultsSize=8 # the size of the results
rankingFunction=tfidf # tfidf, bm25, dfr, ib, lmd (LM Dirichlet), lmj (LM Jelinek-Mercer)

QUERY REDUCTION SETTINGS
queryReduction=true # turn on query reduction [true/false]
QRPercentileNorm=10 # reduction percentile for the T3 layer [0, 100]
QRPercentileT2=10 # reduction percentile for the T2 layer [0, 100]
QRPercentileT1=10 # reduction percentile for the T1 layer [0, 100]
QRPercentileOrig=10 # reduction percentile for the T1 layer [0, 100]
normBoost=4 # boosting for T3 layer
t2Boost=4 # boosting for T2 layer
t1Boost=4 # boosting for T1 layer
origBoost=1 # boosting for T0 layer
ignoreQueryClones=true # ignore the query clones

LICENSE EXTRACTION
license=false # extract license [true, false]
licenseExtractor=regexp # license extractor [ninka, regexp]

EXPERIMENT CONFIGURATIONS
ONLY USED FOR THE EXPERIMENTS OF SIAMESE
similarityMode=tfidf_text_both # elasticsearch similarity function + ngram + normalisation [or both]
cloneClusterFile=soco # prefix of the clone cluster file name [cloplag/soco]
errorMeasure=map # IR error measure [arp/map]
deleteIndexAfterUse=false # delete the index after every run?

SIMILARITY
computeSimilarity=tokenratio # compute similarity of the results
[fuzzywuzzy, tokenratio, none]
simThreshold=50%,60%,70%,80% # the similarity threshold for the four representations
[T1,T2,T3,T4] respectively
github=false # GitHub indexing? (automatically add URL)

Appendix B Wappalyzer Top 100 JavaScript Libraries 2023 compared to corpus (Wappalyzer Pty Ltd, 2023b)

Top 100 Libraries	**Websites tracked**	**Market Share**	**Covered by corpus**	**Market Share corpus**
jQuery	5.013.000	14%	yes	14%
core-js	5.012.000	14%	yes	14%
jQuery Migrate	4.441.000	12%	yes	12%
jQuery UI	3.095.000	8%	yes	8%
Swiper	1.520.000	4%	yes	4%
Modernizr	1.462.000	4%	yes	4%
Lodash	1.354.000	4%	yes	4%
OWL Carousel	1.007.000	3%	yes	3%
LazySizes	983.000	3%	yes	3%
Slick	927.000	3%	yes	3%
Underscore.js	884.000	2%	yes	2%
Moment.js	881.000	2%	yes	2%
FancyBox	873.000	2%	yes	2%
Isotope	856.000	2%	maybe	
Select2	843.000	2%	yes	2%
web-vitals	841.000	2%	yes	2%
Lightbox	704.000	2%	maybe	
Boomerang	597.000	2%	yes	2%
Polyfill	509.000	1%	yes	1%
PhotoSwipe	385.000	1%	yes	1%
Clipboard.js	339.000	1%	yes	1%
Hammer.js	327.000	0.9%	yes	0.9%
prettyPhoto	325.000	0.9%	no	
Flickity	319.000	0.9%	yes	0.9%
YUI	317.000	0.9%	no	
AOS	287.000	0.8%	yes	0.8%
Preact	286.000	0.8%	yes	0.8%
DataTables	216.000	0.6%	yes	0.6%
SweetAlert2	160.000	0.4%	yes	0.4%
MobX	140.000	0.4%	yes	0.4%
Axios	134.000	0.4%	yes	0.4%
lit-html	110.000	0.3%	yes	0.3%
Lozad.js	109.000	0.3%	yes	0.3%
Highlight.js	105.000	0.3%	yes	0.3%
Tippy.js	104.000	0.3%	yes	0.3%
Ethers	90.100	0.2%	no	
lit-element	89.600	0.2%	yes	0.2%
Dropzone	87.100	0.2%	yes	0.2%
FingerprintJS	68.800	0.2%	maybe	
SweetAlert	62.400	0.2%	yes	0.2%
Selectize	60.900	0.2%	yes	0.2%
ScrollMagic	60.000	0.2%	yes	0.2%

crypto-js	59.600	0.2%	yes	0.2%
scrollreveal	55.300	0.1%	no	
Dojo	53.700	0.1%	no	
Apollo	53.000	0.1%	no	
Splide	47.700	0.1%	maybe	
Closure Library	43.300	0.1%	no	
script.aculo.us	36.600	0.1%	no	
Loadable-Components	36.200	0.1%	no	
Twitter typeahead.js	34.900	0.1%	no	
Day.js	34.800	0.1%	yes	0.1%
XRegExp	31.700	0.09%	no	
Choices	30.800	0.08%	yes	0.08%
PubSubJS	29.500	0.08%	no	
Glide.js	27.100	0.07%	yes	0.07%
Zepto	26.700	0.07%	yes	0.07%
Tiny Slider	25.900	0.07%	yes	0.07%
SoundManager	24.900	0.07%	no	
math.js	23.800	0.06%	no	
metisMenu	22.600	0.06%	yes	0.06%
Snap.svg	22.300	0.06%	no	
jPlayer	22.100	0.06%	no	
libphonenumber	20.900	0.06%	yes	0.06%
Howler.js	19.100	0.05%	yes	0.05%
LazySizes unveilhooks plugin	18.400	0.05%	no	
Browser-Update.org	17.300	0.05%	no	
jQuery Modal	16.100	0.04%	yes	0.04%
Vuex	14.400	0.04%	yes	0.04%
Marked	12.800	0.03%	yes	0.03%
Fresco	12.700	0.03%	no	
HeadJS	10.900	0.03%	yes	0.03%
Instant.Page	10.700	0.03%	yes	0.03%
Intersection Observer	10.600	0.03%	yes	0.03%
fullPage.js	8.700	0.02%	yes	0.02%
Moment Timezone	8.100	0.02%	yes	0.02%
Quicklink	7.700	0.02%	no	
Immutable.js	7.300	0.02%	no	
Ramda	7.000	0.02%	no	
Microsoft Authentication	6.400	0.02%	no	
InstantClick	6.200	0.02%	no	
Keen-Slider	6.000	0.02%	yes	0.02%
List.js	5.700	0.02%	no	
lite-youtube-embed	5.700	0.02%	no	
Bootstrap Table	5.700	0.02%	yes	0.02%
Laravel Echo	5.000	0.01%	no	
ClientJS	4.900	0.01%	no	
Ziggy	4.600	0.01%	no	
decimal.js	4.500	0.01%	no	

Slimbox 2	4.200	0.01%	no	
Granim.js	3.900	0.01%	no	
SpriteSpin	3.800	0.01%	no	
Lenis	3.700	0.01%	no	
Glider.js	3.700	0.01%	yes	0.01%
JsObservable	3.700	0.01%	no	
JsRender	3.700	0.01%	yes	0.01%
JsViews	3.700	0.01%	no	
Htmx	3.100	0.01%	no	
FilePond	2.700	0.01%	yes	0.01%
qiankun	2.700	0.01%	no	
		88%	**60/100 covered**	**84%**

Appendix C ECMAScript versions detected in corpus

The following table shows recent ECMAScript versions and characteristic language features introduced by them. The list of newly introduced features is not exhaustive but only contains a few samples to indicate the presence of language features and therefore the presence of ECMAScript versions in the corpus. Some examples for each language features that was detected in the corpus are included with the earliest version it was detected in.

ECMAScript version	Newly Introduced Feature	Corpus Examples	available from version
ES6	spread operator (...)	zone.js	0.13.3 only
		vuetify	3.0.0
		vue-tel-input	6.0.0
	Promise	three	1.133.1
		uikit	3.1.2
	const	webc-miam	3.5.14
		three	1.133.1
ES2016	exponential operator (**)	tsparticles	1.35.0
		three	0.133.1
	includes() method	vuetify	0.16.7
		webtorrent	0.102.3
		webrtc-adapter	7.0.0
ES2017	await	tsparticles	1.3.1
		tinymce	3.0.0
	async	tinymce	6.4.0 only
		vuetify	3.0.0
ES2018	finally() method	statsig-js	1.1.0
		webrtc-adapter	7.3.0
	rest parameter syntax in destructuring	vuetify	3.0.0
		vega-lite	4.16.
ES2019	bare catch clause	typescript	5.1.3
		vue-tel-input	8.0.1
	Array flattening	vega-lite	4.0.0
		web3	4.0.1
		vuetify	3.0.0
ES2020	nullish coalescing operator (??)	vuetify	3.0.0
		webtorrent	2.1.0
		vega-lite	5.1.1
	BigInt type	video.js	7.11.6
	matchAll() method	web3	4.0.1

Appendix D AST resulting from applying obfuscation including control flow flattening to the original source code shown in Figure 2, parsed by *acorn*

References

Aiken, A. Moss [Computer software]. Retrieved from https://theory.stanford.edu/~aiken/moss/

Ain, Q. U., Butt, W. H., Anwar, M. W., Azam, F., & Maqbool, B. (2019). A systematic review on code clone detection. *IEEE Access*, *7*, 86121–86144.

Akram, J., Mumtaz, M., & Luo, P. (2020). IBFET: Index-based features extraction technique for scalable code clone detection at file level granularity. *Software: Practice and Experience*, *50*(1), 22–46. https://doi.org/10.1002/spe.2759

Alam, A. I., Roy, P. R., Al-Omari, F., Roy, C. K., Roy, B., & Schneider, K. A. (2023). GPTCloneBench: A comprehensive benchmark of semantic clones and cross-language clones using GPT-3 model and SemanticCloneBench. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 1–13). IEEE. https://doi.org/10.1109/ICSME58846.2023.00013

Alfageh, D., Alhakami, H., Baz, A., Alanazi, E., & Alsubait, T. (2020). Clone Detection Techniques for JavaScript and Language Independence: Review. *International Journal of Advanced Computer Science and Applications*, *11*(4). https://doi.org/10.14569/IJACSA.2020.01104102

Al-Omari, F., Roy, C. K., & Chen, T. (2020). SemanticCloneBench: A Semantic Code Clone Benchmark using Crowd-Source Knowledge. In *2020 IEEE 14th International Workshop on Software Clones (IWSC)* (pp. 57–63). IEEE. https://doi.org/10.1109/IWSC50091.2020.9047643

Amur, Z. H., Hooi, Y. K., Soomro, G. M., Bhanbhro, H., Karyem, S., & Sohu, N. (2023). Unlocking the Potential of Keyword Extraction: The Need for Access to High-Quality Datasets. *Applied Sciences*, *13*(12), 7228. https://doi.org/10.3390/app13127228

Antal, G., Hegedűs, P., Herczeg, Z., Lóki, G., & Ferenc, R. (2023). Is JavaScript Call Graph Extraction Solved Yet? A Comparative Study of Static and Dynamic Tools. *IEEE Access*, *11*, 25266–25284. https://doi.org/10.1109/ACCESS.2023.3255984

Apple. *JetStream 2 Benchmark: JetStream 2.1 is a JavaScript and WebAssembly benchmark suite focused on the most advanced web applications. It rewards browsers that start up quickly, execute code quickly, and run smoothly.* Retrieved from https://browserbench.org/JetStream2.1/in-depth.html

Arshad, S. [Saad], Abid, S., & Shamail, S. (2022). CodeBERT for Code Clone Detection: A Replication Study. In *2022 IEEE 16th International Workshop on Software Clones (IWSC).* Symposium conducted at the meeting of IEEE. Retrieved from https://ieeexplore.ieee.org/abstract/document/9978260?casa_token=85fTOXRnw4YAAAA A:-LQBNQ4-aM91OdhPN-CPJUV94m51qIiRlkph7eVKkhLK68dYJpJ0oG-oOELVas65XvcePsNG

Bannon, S., & et al (2010). ElasticSearch [Computer software]. Elastic: Elastic. Retrieved from https://www.elastic.co/

Bazon, M. (2012). UglifyJS [Computer software]. Retrieved from https://github.com/mishoo/UglifyJS

Bellon, S., Koschke, R., Antoniol, G., Krinke, J., & Merlo, E. (2007). Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, *33*(9), 577–591. https://doi.org/10.1109/TSE.2007.70725

Bredow, A., & Michel, J. (2010). Library Detector for Chrome [Computer software]. Retrieved from https://github.com/johnmichel/Library-Detector-for-Chrome

Cao, H., Peng, Y., Jiang, J., Falleri, J.-R., & Blanc, X. (2017). Automatic identification of

client-side JavaScript libraries in web applications. In S. Y. Shin, D. Shin, & M. Lencastre (Eds.), *Proceedings of the Symposium on Applied Computing* (pp. 670–677). New York, NY, USA: ACM. https://doi.org/10.1145/3019612.3019845

Cheung, W. T., Ryu, S., & Kim, S. [Sunghun] (2016). Development nature matters: An empirical study of code clones in JavaScript applications. *Empirical Software Engineering*, *21*(2), 517–564. https://doi.org/10.1007/s10664-015-9368-6

Collberg, C., Thomborson, C., & Low, D. (1997). *A taxonomy of obfuscating transformations.*

Dahl, R., justjavac, & et al (2018). Deno [Computer software]. Deno Land Inc: Deno Land Inc. Retrieved from https://deno.com/

Derks, C., Strüber, D., & Berger, T. (2023). A benchmark generator framework for evolving variant-rich software. *Journal of Systems and Software*, *203*, 111736. https://doi.org/10.1016/j.jss.2023.111736

Devore-McDonald, B., & Berger, E. D. (2020). Mossad: defeating software plagiarism detection. *Proceedings of the ACM on Programming Languages*, *4*(OOPSLA), 1–28. https://doi.org/10.1145/3428206

Eckhardt, S., & LL.M. Lüttel, S. Intellectual property and AI: The challenges facing copyright law. In *GoingDigital. The online magazine for the changing legal market* (September 2023, pp. 22–24). Retrieved from https://www.deutscheranwaltspiegel.de/goingdigital/artificial-intelligence/intellectual-property-and-ai-32327/

ECMA TC39 Committee. *ECMAScript 2024 Language Specification*. ECMA-262. Retrieved from https://tc39.es/ecma262/

ECMA TC39 Committee (2024). *ECMAScript® 2025 Language Specification*. ECMAScript Language: Lexical Grammar. Retrieved from https://tc39.es/ecma262/#sec-ecmascript-language-lexical-grammar

Einar, L., & Newman, L. js-beautify [Computer software]. Retrieved from https://github.com/beautifier/js-beautify

Endres, J., & Mühleis, N. Verbotener Code: Urheberrechtliche Probleme bei der Programmierung von Software mit Hilfe von KI. In *MMR* (Heft 10, pp. 723–802). Retrieved from https://beck-online.beck.de/Bcid/Y-300-Z-MMR-B-2023-S-725-N-1

ESLint, Acorn, Babel, & Mozilla Foundation. *ESTree*. Retrieved from https://github.com/estree/estree

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., . . . Zhou, M. (2020, February 19). *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. Retrieved from http://arxiv.org/pdf/2002.08155v4

German, D. M., Manabe, Y., & Inoue, K. (2010). ninka [Computer software]. Retrieved from http://turingmachine.org/~dmg/papers/dmg2010ninka.pdf

Godfrey, M. W., & Kapser, C. J. (2021). Sometimes, Cloning Is a Sound Design Decision! In K. Inoue & C. K. Roy (Eds.), *Code Clone Analysis* (pp. 209–223). Singapore: Springer Singapore. https://doi.org/10.1007/978-981-16-1927-4_15

Guerra Lourenço, P. (2023, April 4). Attacking JS engines: Fundamentals for understanding memory corruption crashes. Retrieved from https://www.sidechannel.blog/en/attacking-js-engines/

Hammad, M., Babur, O., Basit, H. A., & van den Brand, M. (2022). Clone-Seeker: Effective Code Clone Search Using Annotations. *IEEE Access*, *10*, 11696–11713. https://doi.org/10.1109/ACCESS.2022.3145686

Han, S., Ryu, M., Cha, J., & Choi, B. U. (2014). HOTDOL: HTML Obfuscation with Text Distribution to Overlapping Layers. In *2014 IEEE International Conference on Computer*

and Information Technology (pp. 399–404). IEEE. https://doi.org/10.1109/CIT.2014.104

Harder, J., & Göde, N. (2011). Efficiently handling clone data. In J. R. Cordy, K. Inoue, S. Jarzabek, & R. Koschke (Eds.), *Proceedings of the 5th International Workshop on Software Clones* (pp. 81–82). New York, NY, USA: ACM. https://doi.org/10.1145/1985404.1985427

Haverbeke, M. (2019). *Eloquent JavaScript: A modern introduction to programming* (Third edition). San Francisco: No Starch Press. Retrieved from https://eloquentjavascript.net/

Hendrick, S. (January 2022). *Software Bill of Materials (SBOM) and Cybersecurity Readiness*. foreword by Jim Zemlin.

Huang, W., Meng, G., Lin, C., Yan, Q., Chen, K., & Ma, Z. (2023). Are our clone detectors good enough? An empirical study of code effects by obfuscation. *Cybersecurity*, *6*(1). https://doi.org/10.1186/s42400-023-00148-x

Jensen, S. H., Madsen, M., & Møller, A. (2011). Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In T. Gyimóthy & A. Zeller (Eds.), *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (pp. 59–69). New York, NY, USA: ACM. https://doi.org/10.1145/2025113.2025125

Jiang, L., Misherghi, G., Su, Z., & Glondu, S. (2007). DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *29th International Conference on Software Engineering (ICSE'07)* (pp. 96–105). IEEE. https://doi.org/10.1109/ICSE.2007.30

Justjavac (2020). Library Sniffer [Computer software]. Retrieved from https://chromewebstore.google.com/detail/library-sniffer/fhhdlnnepfjhlhilgmeepgkhjmhhhjkh?hl=de

Kachalov, T. (2016). javascript-obfuscator [Computer software]. Retrieved from https://github.com/javascript-obfuscator/javascript-obfuscator

Karabiber, F. TF-IDF — Term Frequency-Inverse Document Frequency. Retrieved from https://www.learndatasci.com/glossary/tf-idf-term-frequency-inverse-document-frequency

Karp, R. M., & Rabin, M. O. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, *31*(2), 249–260.

Kiers, B., & Kochurkin, I. ANTLR [Computer software]. Retrieved from https://github.com/antlr/grammars-v4/blob/master/javascript/

Kim, S. [Seulbae], Woo, S., Lee, H. [Heejo], & Oh, H. (2017). VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy (SP)* (pp. 595–614). IEEE. https://doi.org/10.1109/SP.2017.62

Kirkman, R., Davis, T., Cowley, M., Sauleau, S., & Caslin, T. (2011). cdnjs [Computer software]. Cloudflare: Cloudflare. Retrieved from https://cdnjs.com/api

Koppers, T., Ewald, J., Larkin, S. T., & Kluskens, K. (2016). webpack [Computer software]. Retrieved from https://webpack.js.org/

Kucherenko, A. (2019). JSCPD [Computer software]. Retrieved from https://github.com/kucherenko/jscpd

Kumaran, G., & Carvalho, V. R. (2009). Reducing long queries using query quality predictors. In J. Allan, J. Aslam, M. Sanderson, C. Zhai, & J. Zobel (Eds.), *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval* (pp. 564–571). New York, NY, USA: ACM. https://doi.org/10.1145/1571941.1572038

Lauinger, T., Chaabane, A., Arshad, S. [Sajjad], Robertson, W., Wilson, C., & Kirda, E. (2017). Thou Shalt Not Depend on Me: Analysing the Use of Outdated JavaScript Libraries on the Web. Advance online publication. https://doi.org/10.14722/ndss.2017.23414

Le, H., Fallace, F., & Barlet-Ros, P. (2017). Towards accurate detection of obfuscated web tracking. In *2017 IEEE International Workshop on Measurement and Networking (M&N).* Symposium conducted at the meeting of IEEE.

Lee, H. [Hongki], Won, S., Jin, J., Cho, J., & Ryu, S. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *FOOL 2012: 19th International Workshop on Foundations of Object-Oriented Languages* (p. 96). Citeseer.

The Linux Foundation. *SPDX*. (ISO/IEC 5962:2021).

Liu, X., & Ziarek, L. (2023). PTDETECTOR: An Automated JavaScript Front-end Library Detector. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 649–660). IEEE. https://doi.org/10.1109/ASE56229.2023.00049

Lončarević, S., Skendrović, B., Kovačević, I., & Groš, S. (2023). Detecting JavaScript libraries using identifiers and hashes. In *2023 46th MIPRO ICT and Electronics Convention (MIPRO)* (pp. 1246–1251). IEEE. https://doi.org/10.23919/MIPRO57284.2023.10159971

Mahlpohl, G. JPlag [Computer software]. Karlsruhe Institute of Technology (KIT): Karlsruhe Institute of Technology (KIT). Retrieved from https://github.com/jplag/JPlag

Mapbox, Inc. (2023). *mapbox-gl-js LICENSE.txt*. Retrieved from https://github.com/mapbox/mapbox-gl-js/blob/main/LICENSE.txt

(2023). *maplibre-gl-js LICENSE.txt*. Retrieved from https://github.com/maplibre/maplibre-gl-js/blob/main/LICENSE.txt

McKenzie, S., & et al (2018). Babel [Computer software]. Retrieved from https://babeljs.io/docs/

Misu, M. R. H., & Satter, A. (2022). An exploratory study of analyzing JavaScript online code clones. In A. Rastogi, R. Tufano, G. Bavota, V. Arnaoudova, & S. Haiduc (Eds.), *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension* (pp. 94–98). New York, NY, USA: ACM. https://doi.org/10.1145/3524610.3528390

Mitropoulos, D., Louridas, P., Salis, V., & Spinellis, D. (2019). Time Present and Time Past: Analyzing the Evolution of JavaScript Code in the Wild. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)* (pp. 126–137). IEEE. https://doi.org/10.1109/MSR.2019.00029

Monden, A., Nakae, D., Kamiya, T., Sato, S., & Matsumoto, K. (2002). Software quality analysis by code clones in industrial legacy software. In *Proceedings Eighth IEEE Symposium on Software Metrics* (pp. 87–94). IEEE Comput. Soc. https://doi.org/10.1109/METRIC.2002.1011328

Mozilla Foundation. SpiderMonkey [Computer software]. Retrieved from https://searchfox.org/mozilla-central/source/js/src

Executive Order: Improving the Nation's Cybersecurity (2021).

Ngan, R., Konkimalla, S., & Shafiq, Z. (2022). *Nowhere to Hide: Detecting Obfuscated Fingerprinting Scripts*. arXiv. https://doi.org/10.48550/arXiv.2206.13599

Nichols, L., Dewey, K., Emre, M., Chen, S., & Hardekopf, B. (2019). Syntax-based Improvements to Plagiarism Detectors and their Evaluations. In B. Scharlau, R. McDermott, A. Pears, & M. Sabin (Eds.), *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 555–561). New York, NY, USA: ACM. https://doi.org/10.1145/3304221.3319789

Nicolini, T., Hora, A., & Figueiredo, E. (2024). On the Usage of New JavaScript Features Through Transpilers: The Babel Case. *IEEE Software*, *41*(1), 105–112. https://doi.org/10.1109/MS.2023.3243858

Nikiforakis, N., Invernizzi, L., Kapravelos, A., van Acker, S., Joosen, W., Kruegel, C., . . .

Vigna, G. (2012). You are what you include. In T. Yu, G. Danezis, & V. Gligor (Eds.), *Proceedings of the 2012 ACM conference on Computer and communications security* (pp. 736–747). New York, NY, USA: ACM. https://doi.org/10.1145/2382196.2382274

L'esprit libre (2024, February 16). *Non-respect de la licence GPL: Orange condamné en appel: Open Source : La cour d'appel de Paris a condamné Orange à payer 650.000 euros à la société coopérative Entr'Ouvert pour ne pas avoir respecté la licence GNU GPL v2.* [Press release]. Retrieved from https://www.zdnet.fr/blogs/l-esprit-libre/non-respect-de-la-licence-gpl-orange-condamne-en-appel-39964312.htm

Npm, Inc. (2009). *NPM*. Retrieved from https://www.npmjs.com/

Npm, Inc. (2024, January 22). NPM CLI/Configuring/package.json. Retrieved from https://docs.npmjs.com/cli/v10/configuring-npm/package-json

Oftedal, E. (2018). Retire.js [Computer software]. Retrieved from https://github.com/RetireJS/retire.js

OpenJS Foundation. Node.js [Computer software]. Retrieved from https://nodejs.org/en

OWASP Foundation. *CycloneDX.*

OWASP Foundation (2021). OWASP Top 10 Client-Side Security Risks. Retrieved from https://owasp.org/www-project-top-10-client-side-security-risks/

Pagon, V., Skendrovć, B., Kovačević, I., & Groš, S. (2023). JavaScript Library Version Detection. In *2023 46th MIPRO ICT and Electronics Convention (MIPRO)* (pp. 1240–1245). IEEE. https://doi.org/10.23919/MIPRO57284.2023.10159725

Parr, T., & et al. ANTLR (ANother Tool for Language Recognition) [Computer software]. Retrieved from https://github.com/antlr/antlr4

Pike, R., & Loki. Sherlock [Computer software]. Retrieved from https://github.com/diogocabral/sherlock

Pinckney, D., Cassano, F., Guha, A., & Bell, J. (2023). npm-follower: A Complete Dataset Tracking the NPM Ecosystem. In S. Chandra, K. Blincoe, & P. Tonella (Eds.), *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 2132–2136). New York, NY, USA: ACM. https://doi.org/10.1145/3611643.3613094

Preston-Werner, T. SemVer: Semantic Versioning. Retrieved from https://semver.org/

Princeton University Department of Computer Science (2022). *COS 320: Compiling Techniques: Spring 2022*. Retrieved from https://www.cs.princeton.edu/courses/archive/spring22/cos320/lectures/parsing1.pdf

Ragkhitwetsagul, C., & Krinke, J. (2019). Siamese: scalable and incremental code clone search via multiple code representations. *Empirical Software Engineering*, *24*(4), 2236–2284. https://doi.org/10.1007/s10664-019-09697-7

Richards, G., Gal, A., Eich, B., & Vitek, J. (2011). Automated construction of JavaScript benchmarks. In C. V. Lopes & K. Fisher (Eds.), *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications* (pp. 677–694). New York, NY, USA: ACM. https://doi.org/10.1145/2048066.2048119

Rosen, L. (2004). *Open source licensing: Software freedom and intellectual property law*. Upper Saddle River, NJ: Prentice Hall PTR.

Rossberg, A. (2019). WebAssembly [Computer software]. W3C: W3C. Retrieved from https://webassembly.org/

Rozi, M. F., Kim, S. [Sangwook], & Ozawa, S. (2020). Deep Neural Networks for Malicious JavaScript Detection Using Bytecode Sequences. In *2020 International Joint Conference on Neural Networks (IJCNN)* (pp. 1–8). IEEE.

https://doi.org/10.1109/IJCNN48605.2020.9207134

Runwal, A. N., & Waghmare, A. D. (2017). Code clone detection based on logical similarity: A review. *International Journal of Advanced Research in Computer and Communication Engineering, ISSN*, 2278-1021.

Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K., & Lopes, C. V. [Cristina V.] (2015). SourcererCC: Scaling Code Clone Detection to Big Code. Advance online publication. https://doi.org/10.48550/arXiv.1512.06448

Santos, F., & Vicente, R. (2018). terser [Computer software]. Retrieved from https://terser.org/

Santos, F., & Vicente, R. (2023). *Terser Docs*. Retrieved from https://terser.org/docs/options/

Schleimer, S., Wilkerson, D. S., & Aiken, A. (2003). Winnowing. In Z. Ives, Y. Papakonstantinou, & A. Halevy (Eds.), *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (pp. 76–85). New York, NY, USA: ACM. https://doi.org/10.1145/872757.872770

Sham, S. (2023). The Top 11 Open-Source SBOM tools. Retrieved from https://www.wiz.io/academy/top-open-source-sbom-tools#open-source-sbom-tools-18

Shrestha, S., Shakya, S., & Gautam, S. (2023). Winnowing vs Extended-Winnowing: A Comparative Analysis of Plagiarism Detection Algorithms. *Journal of Trends in Computer Science and Smart Technology*, *5*(3), 213–232. https://doi.org/10.36548/jtcsst.2023.3.001

Sim, S. E., Easterbrook, S., & Holt, R. C. (2003). Using benchmarking to advance research: a challenge to software engineering. In *25th International Conference on Software Engineering, 2003. Proceedings* (pp. 74–83). IEEE. https://doi.org/10.1109/ICSE.2003.1201189

Sinclair, J. (1991). Corpus, concordance, collocation. *Oxford University Press Google Schola*, *2*, 1–10.

Skolka, P., Staicu, C.-A., & Pradel, M. (2019). Anything to Hide? Studying Minified and Obfuscated Code in the Web. In L. Liu & R. White (Eds.), *The World Wide Web Conference* (pp. 1735–1746). New York, NY, USA: ACM. https://doi.org/10.1145/3308558.3313752

St. Jules, D. (2014). JsInspect [Computer software]. Retrieved from https://github.com/danielstjules/jsinspect

Stallmann, R. (1985). *The GNU Manifesto*. Retrieved from https://www.gnu.org/gnu/manifesto.en.html

Svajlenko, J., Islam, J. F., Keivanloo, I., Roy, C. K., & Mia, M. M. (2014). Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *2014 IEEE International Conference on Software Maintenance and Evolution* (pp. 476–480). IEEE. https://doi.org/10.1109/ICSME.2014.77

Vázquez, H. C., Bergel, A., Vidal, S., Díaz Pace, J. A., & Marcos, C. (2019). Slimming javascript applications: An approach for removing unused functions from javascript libraries. *Information and Software Technology*, *107*, 18–29. https://doi.org/10.1016/j.infsof.2018.10.009

Vislavski, T., Rakic, G., Cardozo, N., & Budimac, Z. (2018). LICCA: A tool for cross-language clone detection. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 512–516). IEEE. https://doi.org/10.1109/SANER.2018.8330250

Walker, A., Cerny, T., & Song, E. (2020). Open-source tools and benchmarks for code-clone detection. *ACM SIGAPP Applied Computing Review*, *19*(4), 28–39. https://doi.org/10.1145/3381307.3381310

Wan, B., Dong, S., Zhou, J., & Qian, Y. (2023). SJBCD: A Java Code Clone Detection

Method Based on Bytecode Using Siamese Neural Network. *Applied Sciences*, *13*(17), 9580. https://doi.org/10.3390/app13179580

Wang, J., & Dong, Y. (2020). Measurement of Text Similarity: A Survey. *Information*, *11*(9), 421. https://doi.org/10.3390/info11090421

Wang, T., Harman, M., Jia, Y., & Krinke, J. (2013). Searching for better configurations: a rigorous approach to clone evaluation. In B. Meyer, L. Baresi, & M. Mezini (Eds.), *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (pp. 455–465). New York, NY, USA: ACM. https://doi.org/10.1145/2491411.2491420

Wappalyzer Pty Ltd (2023a). Wappalyzer (Version 6.10.67) [Computer software]. Retrieved from https://chromewebstore.google.com/detail/wappalyzer-technology-pro/gppongmhjkpfnbhagpmjfkannfbllamg

Wappalyzer Pty Ltd (2023b, December 7). JavaScript libraries technologies market share: These are the top JavaScript libraries technologies based on market share in 2023. Retrieved from https://web.archive.org/web/20231207115852/https://www.wappalyzer.com/technologies/javascript-libraries/

WebKit. *SunSpider JavaScript Benchmark*. Retrieved from https://github.com/WebKit/WebKit/tree/main/PerformanceTests/SunSpider

WHATWG (2024, February 27). HTML specification. Retrieved from https://html.spec.whatwg.org/commit-snapshots/bcb3b9a6d989eac9b43cbe06fa0e4463e01034b8/

Wu, Y., Manabe, Y., Kanda, T., German, D. M., & Inoue, K. (2015). A Method to Detect License Inconsistencies in Large-Scale Open Source Projects. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories* (pp. 324–333). IEEE. https://doi.org/10.1109/MSR.2015.37

Yarn [Computer software] (2016). Retrieved from https://yarnpkg.com/

Yu, D., Yang, J., Chen, X., & Chen, J. (2019). Detecting Java Code Clones Based on Bytecode Sequence Alignment. *IEEE Access*, *7*, 22421–22433. https://doi.org/10.1109/ACCESS.2019.2898411

Zakai, A., Dawborn, T., Shawabkeh, M., & et al (2015). Emscripten [Computer software]. Retrieved from https://emscripten.org/docs/introducing_emscripten/about_emscripten.html

Zakeri-Nasrabadi, M., Parsa, S., Ramezani, M., Roy, C., & Ekhtiarzadeh, M. (2023). A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. Advance online publication. https://doi.org/10.1016/j.jss.2023.111796

Zammetti, F. (Ed.) (2022). *Modern Full-Stack Development*. Berkeley, CA: Apress. https://doi.org/10.1007/978-1-4842-8811-5

Zaytsev, J., Pushkarev, D., & et al (2023, November 22). ECMAScript Compatability Table. Retrieved from https://compat-table.github.io/compat-table/es2016plus/

Zou, Y., Ban, B., Xue, Y., & Xu, Y. (2020). CCGraph. In J. Grundy, C. Le Goues, & D. Lo (Eds.), *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (pp. 931–942). New York, NY, USA: ACM. https://doi.org/10.1145/3324884.3416541