Measuring Code Modernity of Codebases Written in JavaScript

THIJS BEUMER, University of Twente, The Netherlands

JavaScript, one of the most widely used programming languages, has undergone significant evolution through various versions of its underlying standard, ECMAScript. This evolution has introduced new features and syntactic changes, which are adopted by developers at varying rates. In this research, we present a method for measuring the "modernity" of JavaScript codebases, by analyzing the features used from different ECMAScript versions. We recall the concept of a modernity signature, which quantifies the relative adoption of new language features within a codebase, and define its meaning in the context of JavaScript. Using static code analysis, we develop a tool that generates modernity signatures for existing JavaScript projects. By normalizing and visualizing the evolution of these signatures over time, we gain valuable insights into the development practices employed by JavaScript developers. The findings of this research contribute to a deeper understanding of how JavaScript codebases evolve in response to changing language features and developer choices.

Additional Key Words and Phrases: JavaScript, ECMAScript, Modernity, Static Analysis

1 INTRODUCTION

JavaScript is a dominant language for web development, with over 98% of websites using it as of November 2024 [22]. Its underlying standard, ECMAScript [13], has been evolving through many versions, introducing new features and syntax that provide developers with more tools to build their applications upon.

Despite the constant development happening on the ECMAScript standard, not all JavaScript codebases immediately adopt newly available features. The rate at which new features are integrated and the relative usage of features from different ECMAScript versions varies significantly between projects. The level of adoption, along with the usage distribution of features from different versions of ECMAScript, forms what we will refer to as the 'modernity' of a codebase. Code modernity refers to the extent to which a codebase leverages features introduced in newer versions of the language, providing insights into feature adoption patterns, development practices, and overall evolution of JavaScript projects.

Investigating modernity is particularly valuable as the relative adoption of newer ECMAScript features, and the evolution of these usage patterns over time, can serve as indicators of active maintenance and code health [1]. In previous research, active maintenance has been correlated to software quality [14], enhanced reliability [16], and a reduction in security vulnerabilities [12]. By extension, analyzing the modernity of a codebase may provide indirect insight into these metrics, offering a novel way of determining the state and evolution of JavaScript projects.

TScIT 42, January 31, 2025, Enschede, The Netherlands

The modernity signature, a concept defined in previous research [1], can capture the evolution of a codebase by mapping the features used in a codebase to their respective ECMAScript versions. A JavaScript function written in ECMAScript version 15 (ES15) may look different from one written in ES6, even when serving the same purpose. Such differences, while not always visible during code execution, may have impact on maintainability, readability, and adoption of best practices within the JavaScript development community.

This paper explores the modernity of JavaScript codebases by defining and measuring the evolution of their modernity signatures over time. A static code analysis tool will be developed to calculate these signatures for various JavaScript projects throughout their lifetimes. The tool will parse JavaScript files, analyze their Abstract Syntax Trees (ASTs), and identify which features from which ECMAScript versions are being used. By visualizing these signatures over time, valuable insights into the evolution of JavaScript codebases can be obtained.

To structure our study, the following research question is addressed:

"To what extent can we use static code analysis methods to reliably detect the modernity of a JavaScript codebase?"

This goal is further narrowed down to the following sub-questions:

RQ1 *How can a modernity signature be defined within the context of the JavaScript language?*
RQ2 *What methods can be used to determine a modernity signature for any give JavaScript codebase?*
RQ3 *What approaches can be used to effectively normalize and visualize modernity signatures over time to reveal meaningful patterns and insights?*
RQ4 *What insights about quality, maintainability, and development practices can be derived solely from the modernity signature of a JavaScript codebase?*

In this paper we will refer to ECMAScript versions as "language versions" and features within the ECMAScript specification as "language features".

After going over related work in Section 2, we will go over the methodology in Section 3 while answering **RQ1** and **RQ2**. Next, Section 4 focuses on running an experiment. Finally, in Section 5 we discuss the results generated during the experiment and answer **RQ3** and **RQ4**.

2 RELATED WORK

Previous research on code modernity has been conducted for several programming languages, including PHP [20], Python [2], C# [19], and Rust [5]. These studies have investigated usage of features from different versions, and patterns observed during the lifetime of codebases written in these languages. Additionally, research on normalization techniques for modernity signatures [24], as well as a study bringing this and the Python and PHP papers together [1], have laid the groundwork for our research.

Although significant work has been done in these languages, JavaScript remains largely unexplored in the context of code modernity. This study builds upon the existing research by applying the concept of modernity to JavaScript.

Static code analysis of JavaScript codebases has also been the subject of previous research, with one study focusing on separate but related JavaScript modules and investigating how they can be analyzed as one network using these techniques [15]. Another study explored the detection of potential points of failure in asynchronous JavaScript programs, specifically analyzing asynchronous structures [17]. Additionally, an empirical study was conducted on static code analysis tools for detecting vulnerabilities in Node.js codebases [6]. These works illustrate the use of static code analysis, and highlight its potential for uncovering otherwise hidden information about JavaScript codebases.

Looking at the history of JavaScript, the evolution of the language is tightly linked with the ECMAScript standard, which describes the language's syntax, semantics, and features. Studies conducted on political and technical aspects of ECMAScript's development [21, 23] have highlighted the processes that have influenced this standard. The political landscape around ECMAScript has evolved, where earlier versions of ECMAScript were subject to more unpredictable updates. This is partially reflected by usage patterns of features in JavaScript codebases, and in turn visible in their modernity signatures. Earlier versions introduce a larger amount of unique features, compared to more modern versions of the language.

3 METHODOLOGY

In this section, we will discuss what a modernity signature means in the context of JavaScript and this paper, define a method for systematically generating modernity signatures for existing JavaScript codebases, and go over the testing approach used to validate an implementation of the method.

3.1 Defining a Modernity Signature

In line with previous research on modernity signatures [1] we define a modernity signature as an n-tuple, where n is defined as the number of different language versions considered. Each entry in the tuple corresponds to a specific language version, and represents the extent to which features introduced in that version are used within the codebase.

Since the majority of features, detectable from the Abstract Syntax Tree (AST) of a JavaScript project, were introduced in language versions 3 and 6 [18], we expect those versions to show significantly higher values within the signature compared to others. To address this bias, we will define two types of signatures.

First, we define the **aggregate** signature, where each entry in the n-tuple reflects the total number of times features, introduced in the corresponding language version, were used in the codebase.

Secondly, we define the **boolean** signature, which counts the number of unique language features used, disregarding the total amount of times they appear.

To illustrate, consider a hypothetical example, with language version V and language features A and B. If for a given codebase we detect A used twice and B used four times, The aggregate signature will be:

$$agg = 2A + 4B = 6$$

The boolean signature, however, will simply count whether a feature appears or not, resulting in a value of:

$$bool = 1A + 1B = 2$$

During the visualization of the results we will apply various, previously researched [24], normalization methods on these two signature types. We do this to mitigate the bias towards language versions 3 and 6 that will likely dominate the signatures. This approach is discussed in detail in Section 4.3.2.

3.2 Static Analysis

To generate aggregate and boolean signatures for any JavaScript codebase, we develop a static analysis tool that parses JavaScript files into ASTs and analyzes each node to detect language features.

3.2.1 Environment. The tool is implemented using Node.js runtime environment [10], chosen for its native compatibility with the JavaScript language and extensive library ecosystem. We use the espree library [9] from Node Package Manager (npm) [11], the package manager for Node.js, as it enables us to parse JavaScript files into their AST representations.

Not every type of file that can be found in a JavaScript codebase is suitable for our analysis. Non-code documents (e.g. README files, configurations files), malformed JavaScript files, or files carrying the `.js` extension but using other ECMAScript implementations such as TypeScript [1], are excluded. These files either do not contain ECMAScript or require additional steps to parse beyond the scope of this research. To address this, we identify all files with the `.js` extension and attempt to parse each one using espree. Only files that can be successfully parsed are included in the signature generation process.

3.2.2 Language Version Determination. After parsing valid files, the tool traverses the ASTs to assign a language version to each node. Since no pre-existing tools provide this functionality in the chosen environment, we implement a custom solution. Our implementation evaluates a language version for an individual node based on its type, content and parent node. To determine what node should receive what version, we use publicly available resources such as, the ECMAScript language specifications [13], community-maintained feature changelogs [18], and a full list of node definitions provided by the eslint-visitor-keys package [7].

Most node types are unique to a single language version. However, certain nodes require additional information to determine their version. An example is the Literal node, which represents values such as booleans, null, undefined, strings and numbers. To determine the appropriate version for the Literal node, we test its content against a regular expression that identifies the specific type of Literal used, and in turn the corresponding language version.

Another notable case is the RestElement node, where the assigned language version depends on the context in which it is used. The use of the RestElement within functions or arrays was introduced

[1]TypeScript is a strongly typed programming language that builds on JavaScript. https://www.typescriptlang.org/

in ES6, while its usage within objects was introduced in ES9. To account for this, we evaluate the parent node of the RestElement to determine the corresponding language version.

3.2.3 Signature Generation. To generate the aggregate signature, we count the total number of nodes assigned to every individual language version. For the boolean signatures, each language feature is assigned a unique identifier, here we count only the presence of unique features per language version, regardless of how often they appear.

3.3 Validating Implementation

Although publicly available sources provide comprehensive information on AST nodes and their corresponding language versions, the available data is vast and intricate. Manually implementing a mechanism to assign versions to individual nodes is prone to human error.

To ensure correctness of the version assignment mechanism, we validate the implementation through a series of tests. For each language feature, we create a JavaScript file containing a 'minimal usage' example and associate it with the corresponding language version. Minimal usage here refers to using the least amount of additional language features to create a valid JavaScript program containing the language feature in question. For any given test file, we now check if the following conditions are met:

(1) *The file can be parsed successfully using language version V.*
(2) *The file cannot be parsed using language version V-1.*
(3) *The modernity signature generated for the file contains at least one detection of language version V.*
(4) *The modernity signature generated for the file contains no detections of any language versions higher than V.*

By validating against these criteria, we ensure that the implementation correctly assigns language versions to nodes when traversing an AST.

4 EXPERIMENT

Our implementation can be found on GitHub as JSModernity [4]. More information on how to install and use the tool can be found in the README.md file.

In our implementation, we process codebases by analyzing all releases in chronological order. For every release, the source code is downloaded, relevant files are determined, signatures are generated, and the resulting signatures are saved with a timestamp for that release. Releases are chosen as anchor points to capture meaningful changes in the codebase, as releases are individual points chosen by developers themselves that represent actual changes in the code and/or its functionality.

4.1 Environment / Experimental Setup

The experiment is run using Node.js version 22.11 on a Windows 11 Pro (10.0.26100 Build 26100) machine with 64GB of RAM and a 12-threaded AMD Ryzen 5 2600X processor. As the signature generation process itself does not depend on a specific system specifications, the same result should be obtained when running on a different system while using the same Node.js version. However, the time needed to generate a signature may vary depending on the system.

4.2 Corpus selection

Included in the experiment is a set of 100 of the most starred JavaScript repositories on GitHub [8]. This dataset is chosen to get exposure to a large variety of developers, as well as codebases that have existed and evolved over longer periods of time. It should be noted that this dataset is not representative of the entire JavaScript landscape and may introduce bias due to its focus on popular repositories. Covering every available JavaScript codebase is beyond the scope of this research.

Not all 100 codebases are applicable for our experiment. Codebases that do not contain JavaScript files or that never had releases are not considered. Examples include codebases containing JavaScript concepts, style-guides, or job interview questions. After filtering out non-applicable codebases, we are left with a total of 69 codebases that can be analyzed.

4.3 Results

The results of the experiment can be found in the JSModernity GitHub repository as Release v1.0.0 [4]. It includes a full list of result signatures as well as generated plots.

4.3.1 Generation. The signature generation process for the previously determined dataset was run in batches of 10 at a time, mainly to prevent spamming the GitHub API when cloning repositories and fetching releases. Individual signature generation processes took anywhere from 1 to 15 minutes depending on the amount of JavaScript code being processed. Codebases containing more and/or larger JavaScript files, or that have more releases take longer to analyze.

4.3.2 Normalization. As discussed in Section 3.1, the resulting signatures often show a dominance of language versions 3 and 6. To address this, we can apply previously researched normalization techniques within the context of modernity [24]. The normalization method best suited for suppressing the bias towards language versions 3 and 6, is the logarithmic transformation. Specifically, we use the following formula to normalize the aggregate signatures:

$$x_i' = \frac{\ln x_i}{\ln \prod_m^{i=1} x_i}$$

For the boolean signature, we use a different normalization approach. We normalize each entry by dividing the value for a given language version by the maximum number of features detectable for that version. Continuing our example from Section **??**, given language version V introduces three features, namely feature A, B and C. If we detect feature A twice and feature C once, the normalized boolean value for language version V would be calculated as follows:

$$bool' = \frac{1A + 1C}{1A + 1B + 1C} = \frac{2}{3} \approx 0.667$$

Note again that feature A is counted only once, as the boolean signature counts usage of unique language features a single time, regardless of the frequency at which they are used.

This normalization approach results in each entry representing the percentage of language features used relative to the total available features for that version of the language.

5 DISCUSSION

For each analyzed codebase we visualize the modernity signatures as 3D surface plots. In these plots the X and Y axes represent language versions and release dates respectively. The Z-axis indicates the relative abundance of features for each language version.

In the plots, we also include a red line showing the release dates of different language versions. Since language features should not technically be used prior to their introduction, we expect to see no peaks to the right of this line.

In this section, we discuss a selection of the generated plots and discuss what information, about a codebase, can be deduced by inspecting them.

5.1 Gradual Adoption of Modern Features

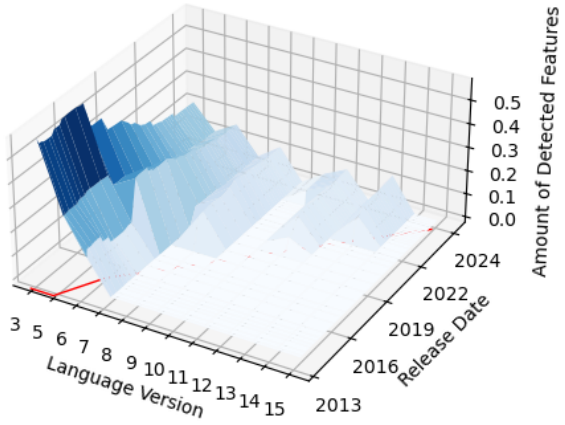

Fig. 1. Aggregate signature evolution for the 'pdf.js' repository

Figure 1 shows the evolution of the aggregate signature for the pdf.js repository by Mozilla. Earlier releases primarily use features from language versions 3 to 6. At the time, these were the only available versions of ECMAScript. Over time, as newer language versions were released (indicated by the red line), features introduced in these newer versions were gradually adopted.

A sudden peak in ES6 feature usage appears between 2016 and 2019. Further investigation reveals that this corresponds to a release from August 2017 where portions of the codebase were refactored to adopt ES6 features, explaining the peak we see.

Figure 2 displays the boolean signatures for the same repository. New information, not visible initially when analyzing the aggregate signature, becomes visible. For example, we see that, since the beginning, a large portion of the total available set of features for language versions 3 and 5 was used. However, for newer versions such as 6, 8 and 11, we see the usage relative to the total available features slowly increasing over time.

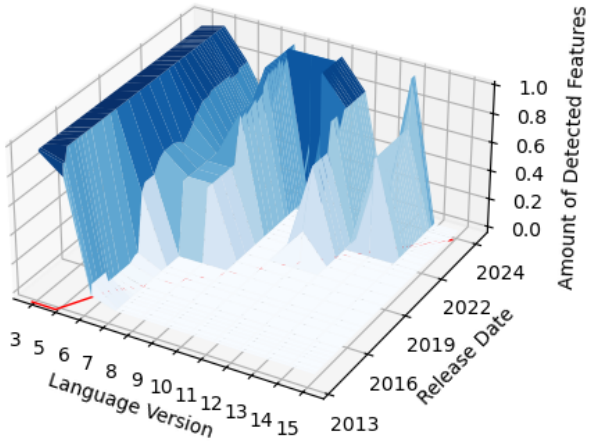

Fig. 2. Boolean signature evolution for the 'pdf.js' repository

5.2 Usage of Experimental Features

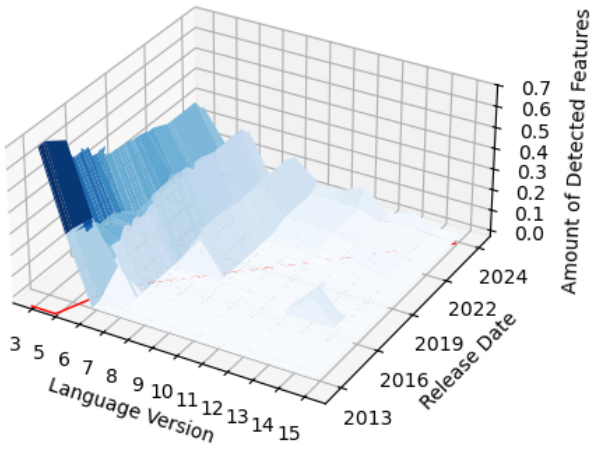

Fig. 3. Aggregate signature evolution for the 'React' repository

The aggregate signature for React, shown in Figure 3, reveals an unexpected peak in ES13 feature usage before its official release. When inspecting the changes in the repository during this period, we see that the developers made use of the static class field syntax before its official inclusion in the ECMAScript standard in 2022. This early adoption is explained by the use of Babel [3], a popular transpiler for ECMAScript.

Transpilers (or source-to-source compilers) are a type of compiler that translate code written in one programming language, into equivalent code for another programming language. In the case of Babel, this is used to convert language features from newer versions of ECMAScript into code for older versions of the standard. This

functionality of Babel allows developers to write code using modern language features while still allowing web browsers or other environments, that take longer to adopt the newest features, to run their code.

In addition to transpiling modern language features, Babel often allows for the transpilation of experimental ECMAScript features, which are candidates for inclusion in upcoming ECMAScript versions. This explains the peak we see for language version 13 in the plot for React.

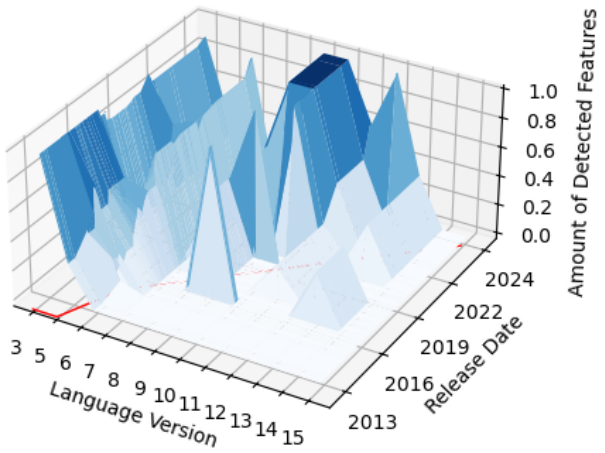

Fig. 4. Boolean signature evolution for the 'React' repository

Looking at the boolean signature for the codebase, as seen in Figure 4, we can see that not only there is an unexpected peak for version 13, there is also a peak for language version 9 before its official release. Again upon further investigation, we see the developers use a rest element within an object. A feature introduced in 2018. However, using Babel, this feature was already available during its experimental phase. The inclusion of experimental features in the codebase suggests that the development team behind React are early adopters of new and experimental language features.

5.3 Minimal Adoption of Modern ECMAScript

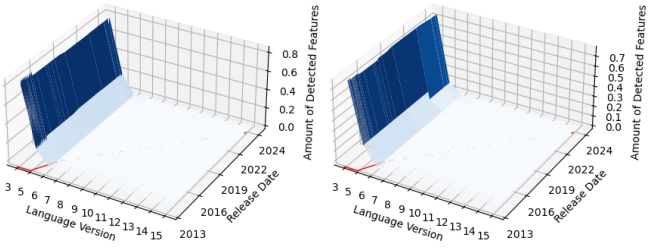

Fig. 5. Aggregate (left) & boolean (right) signature evolutions for the 'ExpressJS' repository

Not all codebases show adoption of newer language features over time. Looking at both the aggregate and boolean plots for ExpressJS (Figure 5), we see that developers are mainly using features from language versions 3 and 5. This suggests that, in contrast to the development team behind React, the ExpressJS developers have chosen a more conservative approach to feature adoption, by sticking to features introduced in earlier versions of ECMAScript.

6 CONCLUSIONS

In this paper we explored the concept of modernity in the context of JavaScript and the standard it is based upon, ECMAScript.

We introduced the meaning of a modernity signature within the context of JavaScript and this paper in Section 3.1, answering **RQ1**. A modernity signature is defined as a tuple that represents the extent to which features from each ECMAScript version are utilized in a given codebase. Given the relative abundance of features in earlier ECMAScript versions, we also differentiated between two types of signatures: aggregate and boolean signatures.

To address **RQ2**, we defined a method for generating modernity signatures for any existing JavaScript codebase, and also outlined our validation process in Sections 3.2 and 3.3. We then applied this method in Section 4 to analyze the top 100 most popular JavaScript repositories on GitHub (based on the total amount of stars received). Of these 100 repositories, we successfully generated 69 signatures, with 31 repositories not meeting our criteria for inclusion in the analysis and thus being excluded.

Finally, in Section 5, we used different normalization techniques and visualized the results by using 3D surface plots, answering **RQ3**. We further discussed the insights gained from these plots. The signatures revealed information about different development practices answering **RQ4**. In certain cases we saw newer language features slowly being used more as the codebase evolved, while others were shown to stick with older language features. We also observed unexpected results where language features were used that would only be introduced into ECMAScript years later. Upon further investigation, we found this to indicate the usage of transpiler libraries like Babel.

7 FUTURE WORK

Building on the work presented in this study, there are several directions for future research that will provide deeper insights into modernity within ECMAScript-based languages.

As highlighted in Section 3.2, our analysis specifically focuses on files with the `.js` extension that contain valid JavaScript. However, the ECMAScript standard has broader implementations than JavaScript alone. One notable example being TypeScript, as briefly touched upon in this paper. Extending our methodology to include other such implementations of ECMAScript will enhance our understanding of the modernity concept within the ECMAScript landscape.

Furthermore, due to the lack of a universally accepted benchmark for selecting JavaScript codebases, our study was limited to the top 100 most popular repositories on GitHub. Although this selection provided valuable insights, it is not representative of the entire set of available JavaScript codebases. Analyzing repositories with different characteristics will likely provide additional insights. Codebases with fewer JavaScript could, for example, show more

sporadic movement in the signature over time, whereas codebases with fewer developers could show unique signatures shaped by the individual preference of those developers for certain features.

Another direction for future work should be, exploring different anchor points beyond the release points used in our analysis. Using different anchor points could reveal alternative patterns. For instance, taking every single commit into account might expose finer details in the evolution of the modernity signature, providing a more continuous view of how codebases evolve over time, especially for smaller, more frequent updates.

Finally, our current method, which only inspects the AST of a JavaScript program, cannot fully capture all features of the ECMAScript standard. For instance, we are unable to differentiate between prototype functions inherently available within ECMAScript and newly defined functions by developers. Expanding the analysis beyond inspecting the AST to detect such constructs, would provide a deeper insight into modernity within ECMAScript.

REFERENCES

[1] C. Admiraal, W. van den Brink, M. Gerhold, V. Zaytsev, and C. Zubcu. 2024. Deriving modernity signatures of codebases with static analysis. *Journal of Systems and Software* 111973 (2024). https://doi.org/10.1016/j.jss.2024.111973

[2] C. P. Admiraal. 2023. Calculating the modernity of popular python projects. https://essay.utwente.nl/94375/

[3] Babel. 2024. Babel transpiler documentation. https://babeljs.io/docs/

[4] T. Beumer. 2025. JSModernity Release v1.0.0. https://github.com/TBeumer/JSModernity

[5] C. Bleeker. 2024. Measuring Code Modernity in Rust. https://essay.utwente.nl/98262/

[6] T. Brito, M. Ferreira, M. Monteiro, P. Lopes, M. Barros, and J. Santos. 2022. Study of JavaScript Static Analysis Tools for Vulnerability Detection in Node.js Packages. (2022). https://doi.org/10.1109/TR.2023.3286301

[7] eslint. 2024. eslint-visitor-keys, Constants and utilities about visitor keys to traverse AST. https://github.com/eslint/js/tree/main/packages/eslint-visitor-keys

[8] EvanLi. 2024. Top 100 Stars in JavaScript. https://github.com/EvanLi/Github-Ranking/blob/master/Top100/JavaScript.md

[9] OpenJS Foundation. 2024. Espree NPM Package. https://www.npmjs.com/package/espree

[10] OpenJS Foundation. 2024. Node.js runtime environment. https://nodejs.org/en

[11] GitHub. 2024. Node Package Manager. https://www.npmjs.com/

[12] A. Ikegami, R. Kula, B. Chinthanet, V. Maeprasart, A. Ouni, T. Ishio, and K. Matsumoto. 2022. On the Use of Refactoring in Security Vulnerability Fixes: An Exploratory Study on Maven Libraries. (2022). https://doi.org/10.48550/arXiv.2205.08116

[13] Ecma International. 2024. ECMAScript language specification version ES2024 / 15th edition, ECMA-262. https://ecma-international.org/publications-and-standards/standards/ecma-262/

[14] B. Lankasena. 2023. Investigating the Impact of Software Maintenance Activities on Software Quality: Case Study. (2023). https://www.researchgate.net/publication/384675050_Investigating_the_Impact_of_Software_Maintenance_Activities_on_Software_Quality_Case_Study

[15] S. Lucz. 2017. Static analysis algorithms for JavaScript. https://ftsrg.mit.bme.hu/thesis-works/pdfs/lucz-soma-bsc.pdf

[16] A. Mateen and M. Akbar. 2016. Estimating software reliability in maintenance phase through ann and statistics. (2016). https://doi.org/10.48550/arXiv.1605.00774

[17] T. Sotiropoulos and B. Livshits. 2019. Static Analysis for Asynchronous JavaScript Programs. (2019). https://doi.org/10.48550/arXiv.1901.03575

[18] Jonna Sudheer. 2024. ECMAScript Features or Cheatsheet. https://github.com/sudheerj/ECMAScript-features

[19] M. Troicins. 2024. Measuring Code Modernity of the C# Language Codebases. https://essay.utwente.nl/101016/

[20] W. van den Brink. 2022. Weighed and found legacy: modernity signatures for PHP systems using static analysis. https://essay.utwente.nl/91794/

[21] J. Vepsäläinen. 2023. ECMAScript – The journey of a programming language from an idea to a standard. (2023). https://doi.org/10.48550/arXiv.2305.01373

[22] W3Techs. 2024. Historical trends in the usage statistics of client-side programming languages for websites. https://w3techs.com/technologies/history_overview/client_side_language/all

[23] A. Wirfs-Brock and B. Eich. 2020. JavaScript: the first 20 years. *Proceedings of the ACM on Programming Languages* 4, HOPL (2020), 77:1–77:189. https://doi.org/10.1145/3386327

[24] C. Zubcu. 2023. Effect of Normalization Techniques on Modernity Signatures in Source Code Analysis. https://essay.utwente.nl/96034/