TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Ali Atakan Basaran 223534IVCM

EVALUATING LARGE LANGUAGE MODELS FOR JAVASCRIPT FILE CLASSIFICATION USING HYBRID ANALYSIS

Master's Thesis

Supervisor: Fuad Budagov
MBA

Tallinn 2025

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Ali Atakan Basaran 223534IVCM

SUURTE KEELEMUDELITE HINDAMINE JAVASCRIPTI FAILIDE KLASSIFITSEERIMISEL HÜBRIIDANALÜÜSI ABIL

Magistritöö

Juhendaja: Fuad Budagov
MBA

Tallinn 2025

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Ali Atakan Basaran

02.01.2025

Abstract

Malicious JavaScript files have been a popular attack vector for both server and client applications for years. Machine learning has been used for malicious JavaScript file detection in the form of static and dynamic analysis, but existing implementations may not be sufficient when it comes to obfuscated JavaScript files. During the last year, the topic of using large language models for static analysis has gained traction. The goal of this research is to explore the applicability and development of a framework capable of classifying JavaScript samples utilizing hybrid analysis with large language models using different prompting methods and evaluating the results across different models to demonstrate a novel method that can be used for mitigating cyber threats, as well as blocking privacy-invading JavaScript files such as trackers with the highest accuracy as possible. The outcome of the research shows a promising success of the proposed approach to JS file detection in the topic, achieving robust results. Future work could involve refining hybrid analysis methodologies and expanding model capabilities to address the limitations observed, ultimately contributing to better cyber threat detection.

The thesis is written in English and is 40 pages long, including 8 chapters, 42 figures, and 2 tables.

Annotatsioon

Suurte keelemudelite hindamine Javascripti failide klassifitseerimisel hübriidanalüüsi abil

Pahatahtlikud JavaScripti failid on olnud populaarne ründevektor nii serveri- kui ka kliendirakendustes juba aastaid. Masinõpet on kasutatud pahatahtlike JavaScripti failide tuvastamiseks läbi staatilise kui ka dünaamilise analüüsi, kuid hetkel olemasolevad lahendused võivad osutuda ebapiisavaks, kui tegemist on hägustatud (obfuscated) JavaScripti failidega. Viimase aasta jooksul on suurenenud huvi suurte keelte mudelite kasutamises ja staatilise analüüsi sooritamiseks tuvastamaks pahatahtlike JavaScripti faile. Selle uurimistöö eesmärk on uurida ja arendada raamistik, mis suudaks klassifitseerida pahatahtlike JavaScripti näidiseid, kasutades hübriidanalüüsi ja suuri keelemudeleid erinevate promptimise meetoditega. Tulemusi hinnatakse erinevate mudelite lõikes, et näidata uudset meetodit, mida saab kasutada küberohtude leevendamiseks ning privaatsust rikkuvate JavaScripti failide, näiteks jälgijate (trackers), blokeerimiseks võimalikult kõrge täpsusega. Uurimistöö tulemused on näidanud pakutud lähenemisviisi edukust JS-failide tuvastamisel, saavutades selgelt nähtavaid tulemusi. Tulevikus võiks antud uurimustöö hõlmata hübriidanalüüsi metoodikate täiendamist ja mudelite võimekuse laiendamist, et lahendada antud uurimistöö raames täheldatud piiranguid ning seeläbi paremini kaasa aidata küberohtude tuvastamisele.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 40 leheküljel, 8 peatükki, 42 joonist, 2 tabelit.

List of Abbreviations and Terms

JS	JavaScript
LLM	Large Language Model
AST	Abstract Syntax Tree
NW	Node Webkit
NPM	Node Package Manager
API	Application Programming Interface
PDF	Portable Document Format
HTML	HyperText Markup Language
WScript	Windows Scripting Host

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Motivation

Over the past years, malicious JavaScript files have become a popular attack surface for both server, client, and web applications. Using machine learning for malware detection has been a topic for a long time and there have been implementations made for JavaScript using static analysis with large language models, however falling short on obfuscated files.

To overcome this shortcoming, as well as increase the support for the architecture of ever-getting complex web applications, hybrid analysis will be utilized for the classification. Hybrid analysis is an analysis method that uses statically available resources such as source code or bytecode combined with the execution result of a given sample. To capture the execution of the samples a sandbox will be used.

A prompt derived from static and hybrid analysis results of the JavaScript sample will be used as input for the large language model. By providing the static and hybrid analysis results in the same context, the rate of false positives during prediction can be reduced.

The motivation for this research is to leverage hybrid analysis of JavaScript files with large language models using the analysis output and classify the purpose of a given file using zero-shot and few-shot prompting methods, then evaluate the metrics across different models with selected indicators.

1.2 Research problem

While machine learning in cybersecurity is already commonly used, the applications of new machine learning breakthroughs such as large language models for the detection of JavaScript files have already become a prominent research focus. Existing approaches are based on static analysis and hybrid analysis which combines static and dynamic analysis that shows promise on other machine learning approaches, its use with large language models is yet to be explored.

The main research problem is exploring the limitations of current and past implementations using LLMs and other machine learning approaches while exploring the hybrid analysis with LLMs and addressing the challenges and limitations in the process such as the

context length, as JavaScript files and the hybrid analysis result can exceed the token limits of most models. This may lead to truncated analysis and reduce classification accuracy. Additionally, JavaScript's obfuscated and minified files can complicate model interpretation due to overlapping features.

1.3 Research goal

The main goals of this research are aiming to;

1. Develop a JavaScript hybrid analysis framework that is usable for files targeting web, server, and desktop applications.

2. Investigating how hybrid analysis results can be used to generate a context representation of JavaScript files.

3. Assess the abilities of LLMs utilizing this representation to overcome challenges and shortcomings of static analysis such as obfuscation.

4. Benchmarking zero-shot and few-shot prompting techniques to see changes in the classification accuracy.

5. Analyze the results address limitations and challenges and explore the future work needed to overcome them.

1.4 Research scope

The main research scope for this topic is to develop a framework for assessing large language models' capabilities in JavaScript file classification using hybrid analysis with the selected classifications and analyze the success rate of different models using different prompting techniques such as few-shot prompting and zero-shot prompting.

1.5 Research Questions

The research questions, which encompass the goal of the thesis, are as follows:

- **RQ1:** Is it possible to use LLMs with hybrid analysis for classification accurately?
- **RQ2:** When using LLMs for classification, are there any accuracy changes when few-shot prompting is used?
- **RQ3:** Will different LLMs result in the same classifications with the same input?

1.6 Novelty

The current state of the literature shows a gap in the hybrid analysis of JavaScript files using large language models that are used for classification. The goal and the importance of this research is to provide a framework to evaluate the large language models for accomplishing the expected task and closing the gap in the research.

2. Literature Review

2.1 Theoretical Background

JavaScript (JS) is an interpreted programming language created by Brendan Eich in 1995 as a scripting language for making web pages interactive. Over the years JS has evolved from a simple scripting system for web pages to a programming language that is used in server-side development, mobile app development, desktop application development, and modern web development [1]. Based on the study in 2020, JS was the most popular programming language used on open-source repositories available on GitHub [2].

In recent years with the advancement of web application technologies, the widespread adaption of JS on the server-side applications, and scripting support for legacy systems still being installed by default for applications like Windows Scripting Host (Wscript) [3] and PDF have expanded the capabilities of JavaScript while expanding the attack surface for the purposes ranging from basic privacy intrusion to various malicious activities like remote code execution or information stealing [4].

One advancement during the same period was the machine learning models that are capable of performing contextual analysis while providing reasoning such as Large Language Models (LLMs), LLMs are machine learning models that are trained on vast number of text data including but not limited to programming code to perform tasks such as summarizing content and answering questions. There were research efforts that combined LLMs with JS in the domain of cybersecurity for vulnerability detection, bug detection, secret finding, and malicious file detection [5].

When the current state of the implementations using LLMs is observed, one thing that was seen is the reliance on static analysis only to perform the task. Static analysis is performed by examining the source code of the file or syntactic features, like the Abstract Syntax Tree (AST) and opcodes, that are generated from it. AST is a structural tree representation of the source code where each node of the tree stands for the construction in the code whether a function calls binary operation or variable assignment, and the opcode is the compiled version of the AST before execution by the JS engine [6]. With the common usage of minification for JS files used by websites, which is a process that changes variable names and code paths to as small as possible to save on bandwidth and some JS files using obfuscation techniques that can behave similarly can cause the static analysis to fall short

on accurately analyze the files. To overcome this, hybrid analysis can be used.

Hybrid analysis is an analysis technique where static and dynamic analysis are conducted, and the dynamic analysis can be run on a sandbox environment that mimics the target system or on the actual target [7]. The dynamic analysis's results reveal the usage of the file by capturing or instrumenting what the obfuscated or minified code is resolved to. To achieve this kind of analysis with the JS files, a runtime environment that can expose all the required application programming interfaces (APIs) for the target environment of the file is needed. The most common runtimes JS files targeting for, namely; web, server-side, PDF, and Wscript are chosen for the sandbox environment to support APIs for [8]. To achieve the goal of exposing these APIs during dynamic analysis with minimal complexity, a runtime that supports both web and server-side APIs, the node-webkit (NW.js) project is selected as the dynamic analysis runtime environment.

NW.js extends the JS environment of a web browser, in this case, Chromium, with the server-side JS runtime, NodeJS which has polyfills for the APIs of Wscript and PDF runtimes available [9]. Polyfilling is the process of reimplementing features that are not natively supported for the current runtime. This approach for dynamic analysis allows both web and other JS files to be executed as accurately as possible in the same environment and keeps the evaluation inputs the same across the files targeting different runtimes [10].

2.2 Search Strategy

Based on the problem statement and research questions, the systematic literature review method, as outlined by Budagov et al. [11] and following the structured guidelines of the Kitchenham method, was applied. Kitchenham method focuses on key steps, including developing a protocol, defining research questions, establishing a search strategy, and ensuring rigorous data extraction and reporting practices. A set of keywords: *"machine learning"*, *"hybrid analysis"*, *"large language models"*, *"malware detection"*, *"dynamic analysis"*, and *"cybersecurity"* in combination with *"JavaScript"* were selected to conduct a systematic literature review using electronic databases such as IEEE Xplore, Web of Science, Scopus and SpringerLink.

A search string combining the selected keywords is generated to include all relevant research. The literature review focused on journal articles and conference papers. The publication period spanned January 2018 to October 2024, highlighting recent developments and reflecting the evolving nature of emerging technologies in machine learning and cybersecurity in the context of JS. A summary of the search strategy is given in the following table 1.

Electronic databases	IEEE Xplore Web of Science Scopus SpringerLink
Type of searched literature	Journal and Conference Papers
Search string	(“machine learning” OR “hybrid analysis” OR “large language models” OR “malware detection” OR “dynamic analysis” OR “cybersecurity”) AND “JavaScript”
Language of the study	English
Publication period	From January 2018 to October 2024

Table 1. Search strategy

2.3 Inclusion and Exclusion Criteria

The inclusion criteria used for the search results were:

1. Papers, articles, and journals published between the years 2018-2024.
2. Literature that include the keywords.
3. Literature that are publicly accessible.
4. Literature that is relevant to the topic and goal of this research.

Exclusion criteria for the search results were:

1. Literature about different programming languages than JavaScript unless it is relevant to the goal of the topic.
2. Literature that is not related to classification or analysis.

2.4 Literature Search and Selection

The literature search resulted in 1191 articles on the sources as follows; IEEE Xplore (n = 377), Web of Science (n = 312), Scopus (n = 182), Springer Link (n = 320). First, the duplicate papers on the result set are merged, making the result set 995 papers. After a preliminary screening based on the titles of the resulting papers, papers with titles that did not fall within the scope of this research were excluded resulting in 183 articles left for screening.

The screening process excluded 152 papers, 64 due to not being related to JavaScript but to other programming languages or platforms, and 88 for not being related to analysis or classification in the context of JavaScript. Lastly, an accessibility check was conducted which excluded 6 papers that are behind a subscription resulting in 25 papers for the literature review. The diagram of the selection process is given in figure 1.

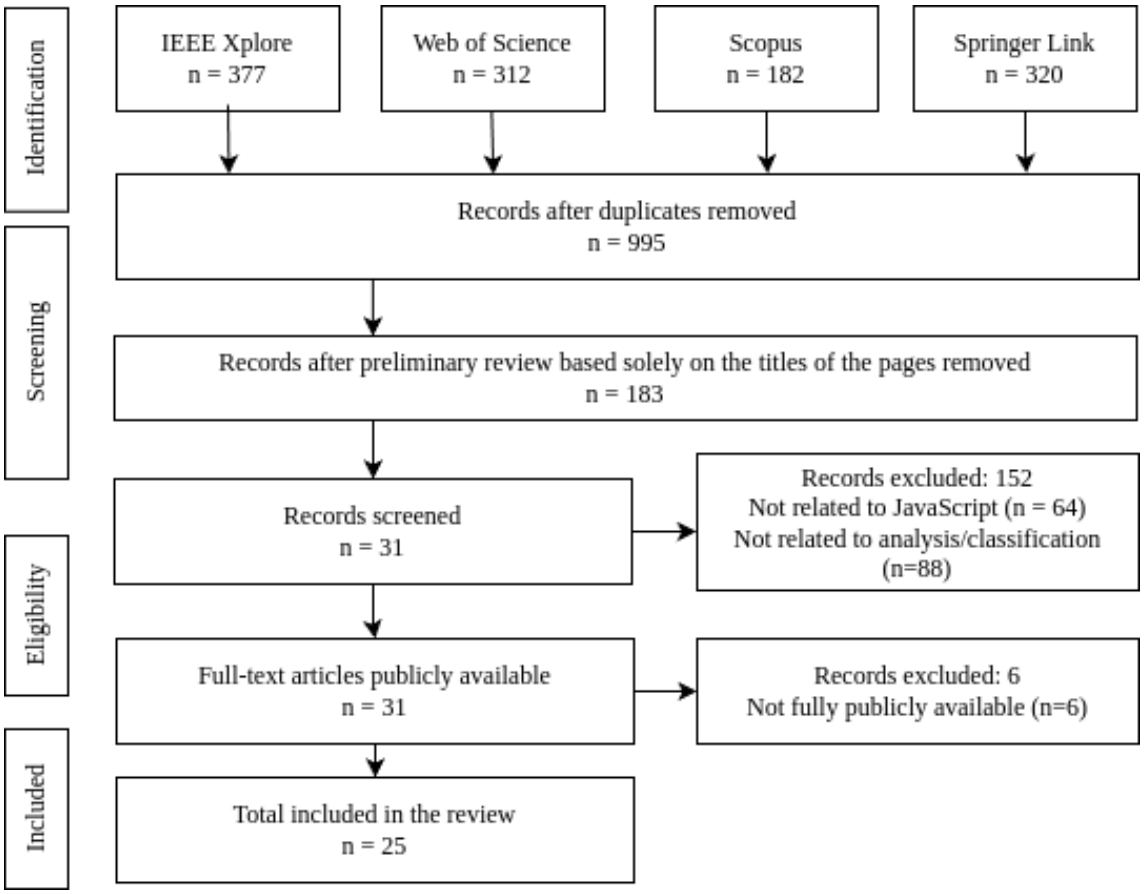

Figure 1. Diagram of the literature selection process

2.5 Summary of the selected literature

The following section reviews the selected literature in the research question's context to identify the research gap and explain the motivation for this research's goal based on the previous research.

JavaScript files have become a popular attack surface for server and client applications. Neural networks and machine learning have been showing promising results when used on the classification of malicious and non-malicious files with the advancement of deep learning methods [12]. One such deep learning method, large language models, and its application in cybersecurity applications became a topic of interest over the past years [5].

Earlier research from Fang et.al. [13] [14] shows the feasibility of using machine learning to accomplish this goal for JavaScript files. The JSContanta [15] shows using large language models for the static analysis of the JavaScript code is also possible.

The use of static analysis is not only limited to malicious file detection. Research from Brito et al., Chan et. al., and Chinthanet et. al. [16] [17] [18] shows that the static analysis with LLMs shows promising results for vulnerability detection in JavaScript code. LLMs' capabilities of analyzing source code for vulnerabilities are also further benchmarked by Purba et.al [19] and Gao et. al. [20] showing the strength of LLMs for this task. However, this approach still has the weakness of the code being obfuscated.

The effects of obstruction during static analysis and solutions using machine learning are further explored in the research by Ren et.al [21]. One approach from the research TransAST [22] reverses the obfuscation using a translation model. The research from Moog et.al [23] and Si et.al [24] shows analyzing the AST of the JavaScript file can also be used to overcome this challenge. Another approach for overcoming the obfuscation from Alazab et.al. [25] is using the opcode from compiled AST for detection. Research from Rozi et.al [26] also shows using machine learning to detect malicious scripts by analyzing the opcode. The research from Lu et.al also provides insights about using compiled AST into opcode for increased accuracy on pattern matching [27].

Static analysis alone is insufficient to address all threats, leading to the adoption of dynamic and hybrid analysis approaches. Sandbox analysis, previously used for executable files, has been adapted to JavaScript. Kishore et al. [28] demonstrated the feasibility of sandboxing JavaScript for hybrid analysis.

Research from He et.al. [29] uses a hybrid approach that combines these methods and focuses on the classification of the JavaScript file. However, this approach does not take into account the contextual analysis provided by large language models. The goal of this research is to use the same hybrid approach with a large language model.

Previous research from Xiao et al. [30] used hybrid analysis to implement a security benchmark suite for server-side JavaScript. When it comes to other hybrid analysis approaches, the research by Koide et.al successfully implements a phishing detector using dynamic analysis on the website HyperText Markup Language (HTML) code and screenshot [31] but without the focus on JavaScript files. Research from Rozi et.al [32] focuses on JavaScript files for website detection but only with static analysis using AST. A dynamic analysis approach that was used in the research of He et.al [33] incorporated a browser extension to monitor the execution of the JavaScript in the webpage. This approach has proven effective for detecting malicious activity in JS files used on web pages.

A more general approach that monitors the execution from the runtime by Jueckstock

et.al [34] also allows dynamic analysis of any JS file. This approach can be used as the theoretical basis of the proposed dynamic analysis system proposed in the background section which also allows the analysis of JS payloads in portable document format (PDF) files which is a prevalent attack surface based on the work of Lemay et. al. [35]

The literature review shows the gap for hybrid analysis of JavaScript files using large language models that are used for classification. While the previous methods have focused on static or dynamic analysis of JS files, the usage of LLMs during hybrid analysis is still unexplored. The goal and the importance of this research is to provide a framework to evaluate the large language models for accomplishing the expected task and closing the gap in the research.

3. Methodology

To achieve the research goal and answer the research questions, an empirical method of testing will be conducted using real JS samples with the inputs obtained.

3.1 Dataset Collection

To accurately evaluate the capabilities of hybrid analysis and LLMs, JS samples that are publicly accessible and up to date need to be obtained. Although there are many types of JS files targeting different environments, the sample collection needs to be scoped to a selected inclusion criteria. This research aims to evaluate the LLMs with the most common threat scenarios. Due to this reason, samples that are websites, PDF files with JS code, Node packages and Wscript files will be collected for the dataset.

3.2 Labeling

With the broad threat landscape and privacy invading JS files, a set of labels with their definitions must be defined to be used for labeling and classification. To keep the research scope on evaluation of the capabilities of hybrid analysis and large language models, four classifications are generated based on the work of Biswal and Pani [36]. These classifications are;

Malware: Any malicious JS code that is explicitly designed to perform malicious actions such as data theft, remote control, ransom or install additional harmful files are labeled as malware.

Adware/Telemetry: JS files that are primarily used for collecting user data, such as browsing habits or personal information and the files that are used for serving advertisement are labeled as adware/telemetry. While these files are not malicious, they can compromise the user privacy.

Skimmer: JS files used to skim payment information from payment forms will be labeled as skimmer. These kinds of files are often found on web pages that are taken over by attackers and injected into the legitimate checkout sections.

Normal: The normal label will be used for JS files that possess no threats or for the files

that are not falling under any other categories. This label will be used for legitimate JS files.

3.3 Evaluation

The evaluation will be conducted by running classification tasks with different LLMs using the labeled dataset. To further benchmark the LLMs, each sample in the dataset will be presented with a few-shot and zero-shot prompt and the results will be analyzed with selected metrics and compared.

4. Hybrid Analysis

This section aims to provide insight into how the hybrid analysis is conducted, going over the static analysis, which information is extracted, and the dynamic analysis with the technical details of the sandbox setup.

4.1 Static Analysis

Static analysis is the process of analyzing a code without executing it. This is achieved by examining the source code or any information it generates. In, JS one such information is the AST which is a structural tree representation of the source code. Each node of the tree represents the construction of the code, whether a function call, binary operate, on, or variable assignment, which will be compiled to a format the execution engine can use.

4.1.1 Source code analysis

Source code analysis involves going through the file source to identify the purpose of the file. With the proven capabilities of LLMs related to source code analysis based on the previous works on vulnerability detection, the source code of the files will be directly used without any additional preprocessing.

4.1.2 AST analysis

JavaScript abstract syntax tree is a representation of the functional part of the source code excluding the additional data such as comments in a structured format. Throughout this research, the AST parsing library esprima is used. The AST representation of the sample code given in figure 2 is represented in appendix 2.

```
1 function answer() {
2     const x = 42
3     console.log(x);
4 }
5 answer();
```

Figure 2. Example of JavaScript code

To briefly demonstrate the functionality of AST nodes, an example node targeting the variable declaration in the code above will be used.

```
{
    "type": "VariableDeclaration",
    "declarations": [
        {
            "type": "VariableDeclarator",
            "id": {
                "type": "Identifier",
                "name": "x",
                "range": [
                    88,
                    89
                ]
            },
            "init": {
                "type": "Literal",
                "value": 42,
                "raw": "42",
                "range": [
                    92,
                    94
                ]
            },
            "range": [
                88,
                94
            ]
        }
    ],
    "kind": "const",
    "range": [
        82,
        94
    ]
}
```

Figure 3. Example AST node

The outer element "type" having the value "VariableDeclaration" indicates this node is a variable declaration statement with the "kind" indicating this is a "const" to specify the variable that will be declared as a constant. This node type can have multiple declarations, in this example, there is only one with the "type" having a value of "VariableDeclarator". The "id" node has the type identifier with the name value "x" which identifies the variable name as "x" and the "init" node has the type "literal" with the integer value of 42, specifying

that this variable will be initialized with the integer value of 42 after declaration.

Based on previous research on literature review related to AST and LLMs, LLMs can analyze the raw AST input directly however, since the "range" objects are specifying the character positions of the nodes in the source code which can be considered not relevant for such analysis. Due to this reason, the range objects will be omitted from the AST before being provided as input to LLMs to save on resources.

4.2 Dynamic analysis

Dynamic analysis is an analysis method where the code is executed in a monitored runtime where the behavior is observed. Due to the malicious samples in the dataset, the dynamic analysis will be conducted in a sandboxed environment. Sandboxing is a security technique that isolates access to networking, filesystem, and memory during execution to prevent potentially malicious files from causing harm to the host system or accessing sensitive resources. In this research, for ease of use and simplicity, containerized sandboxing with Docker is used [37].

4.2.1 Instrumentation

To obtain usable data the JS execution inside the sandbox needs to be monitored. This is achieved by instrumentation. Instrumentation is the practice of adding hooks, logs, or custom code to monitor and analyze the code behavior. In this research's case, a method where selected function calls, variable accesses, and assignments are instrumented is implemented using AST patching. After traversing the AST for the top-level nodes with the node types having the variable definitions, assignments, function calls, and function definitions, the discovered top-level nodes are instrumented by being shifted as a child node under a new top-level node that is inserted to invoke the sandbox's logging function named "sandbox". The diagram of this operation is given on the figure 4. The implementation details of the sandbox function are given in section 4.3.

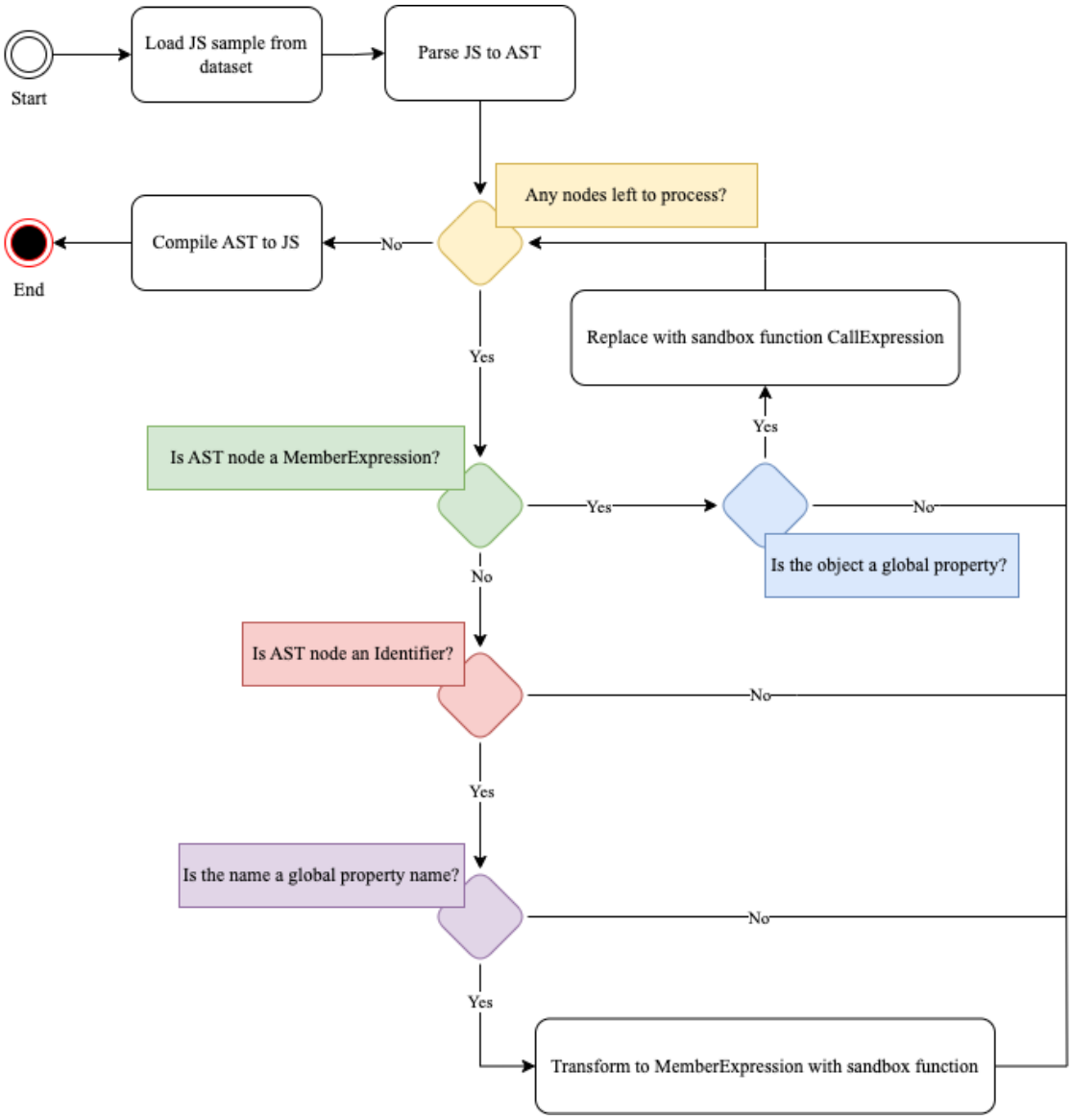

Figure 4. Diagram of the AST patching process

4.2.2 Runtime selection

The JS files planned to be used in the dataset are JS files targeting websites, server-side applications, PDF, and Wscript. Due to this reason a runtime that supports both web and server-side APIs, the node-webkit (NW.js) project is chosen as the dynamic analysis runtime environment.

NW.js extends the JS environment of a web browser, in this case Chromium using, NodeJS, a server-side JS runtime that also allows polyfilling for the Wscript and PDF runtime APIs. With this dynamic analysis method, web and other JS files can be run as correctly as feasible in the same environment while maintaining the same outputs for all the files

regardless of their target runtimes.

Another reason the NW.js is chosen is the ability to be controlled remotely using browser test automation tools while the process is residing inside the sandboxed environment as well as request and runtime modifiers offered by the test automation tools.

Test automation tools allow managing environment code execution and file loading behavior. By using the code execution capabilities, the instrumentation code is executed on the sandbox before any JS file is loaded, and by using the file loading hooks, the AST patching is implemented to instrument the JS files.

4.3 Sandbox function

A function named "sandbox" is injected into the runtime before any code is executed. This function will be used as the sandbox's output function, allowing dynamic analysis on any file that is successfully patched on the AST patching step.

In JavaScript, any function, method, or variable defined in the top-level context is also accessible as a property under a global property named "globalThis". When the sandbox code is injected, all the properties under globalThis are wrapped. This ensures the top-level runtime objects such as "console", "eval" or "window" are instrumented too.

When it comes to the sandbox function that was injected before selected nodes on the AST patching steps is used for monitoring any method or property call or assignment after it. To be used in LLM input, each operation that is captured by the sandbox function is logged with what the operation was, what the property accessed or called and what the value assigned or passed in a format that closely resembles step-by-step execution. Since the research goal is classification, reoccurring events can be omitted from the logs to save resources. A partial sandbox output is given in figure 5 and the diagram of the sandbox function is given in figure 6.

```
...
[main.js] Accessed property: console
[main.js] Accessed property: log
[main.js] Called function: log with arguments: [("hello world")]
[main.js] Accessed property: document
[main.js] Accessed property: createElement
[main.js] Called function: createElement with arguments: ["fieldset"]
[main.js] Accessed property: appendChild
[main.js] Called function: appendChild with arguments: [{"innerHTML":""
...
```

Figure 5. Sample sandbox output

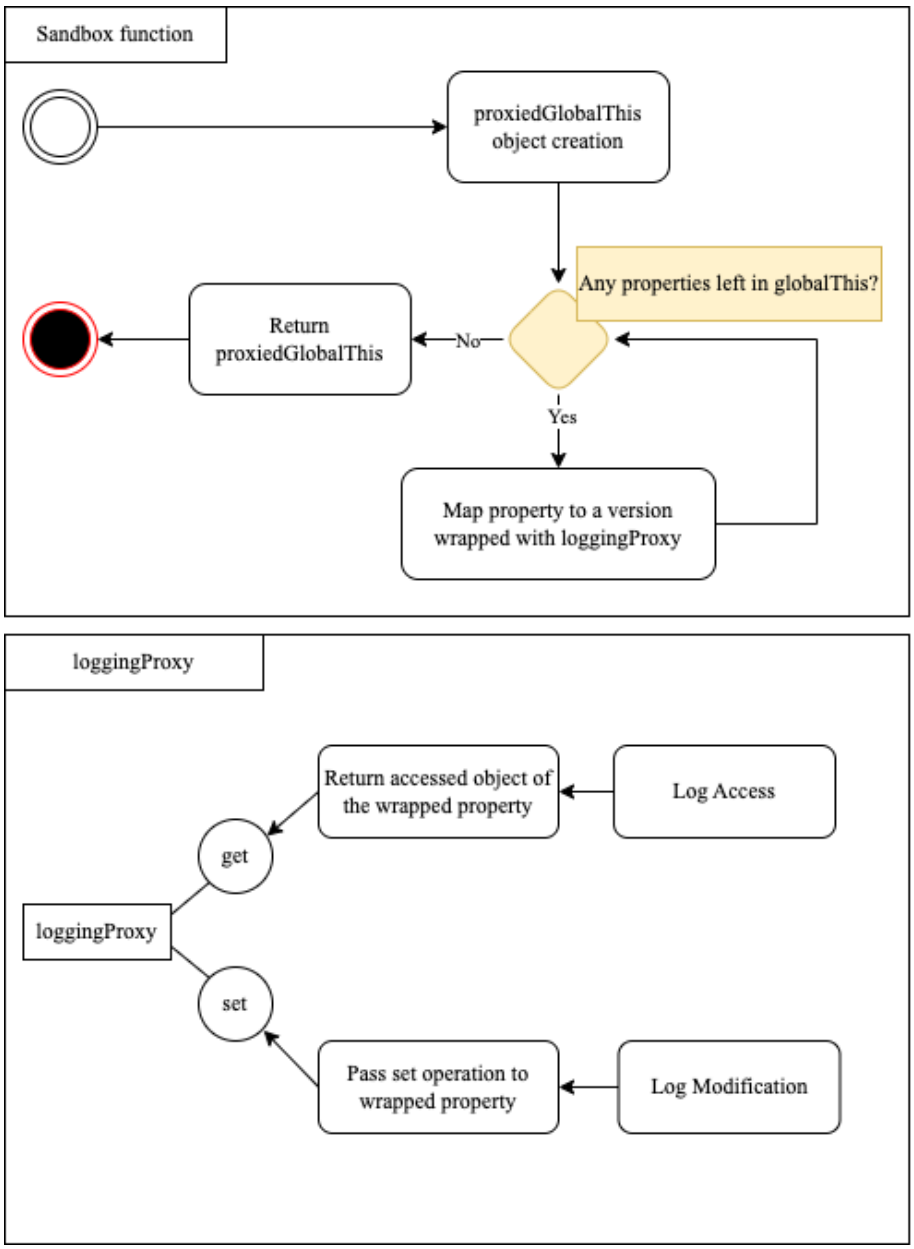

Figure 6. Diagram of the sandbox function

4.4 Framework design

The overall framework is designed as a sandbox that can read the dataset, apply transformations, and automatically run the runtime environment on an isolated Docker task that prevents the running sample from accessing the host system's memory, filesystem, and processes [38]. The network was not isolated in the sandbox for allowing websites in the dataset to function, as well as malicious scripts that are loading secondary payloads to be able to continue executing. The runtime requests are being monitored by the sandbox manager for instrumentation of files loaded from the dataset, these additional requests will also have instrumentation applied. After the sandbox step, the framework also generates the relevant prompt and supplies it to the selected models for classification. The overview diagram of the sandbox framework is given in the figure 7.

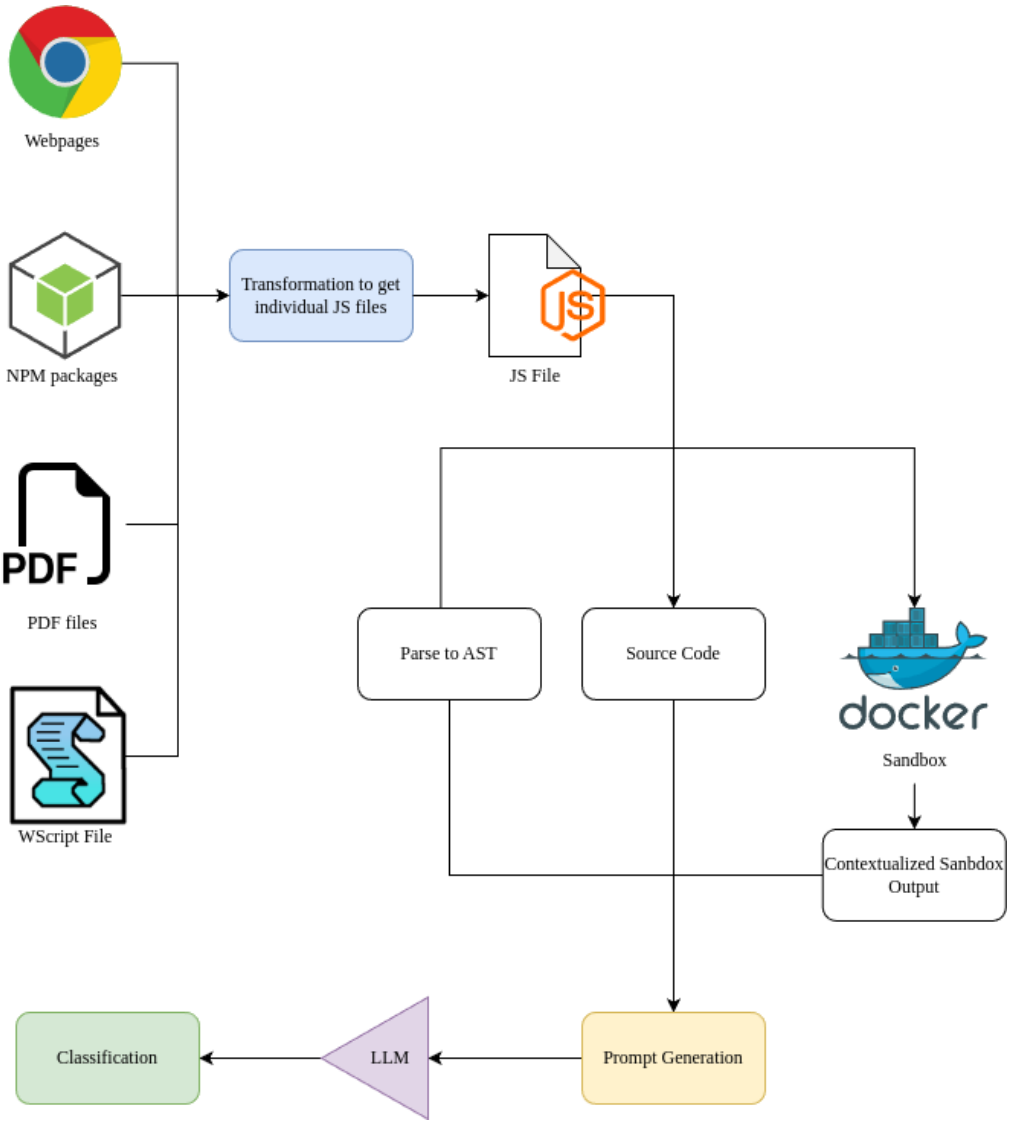

Figure 7. Overview diagram of the framework

5. Evaluation

This section outlines the process of evaluating LLMs for JS file classification. It covers dataset collection, transformation, and labeling, as well as the models selected for testing with the evaluation metrics and prompting techniques that will be used.

5.1 Dataset

This section provides the information related to dataset that will be used in evaluation with the encountered challenges and solutions for these challenges as well as technical details of these solutions.

5.1.1 Dataset collection

For the generation of the dataset to be evaluated in this research, four forms of JavaScript files are targeted. These are the JS files that are standalone and can be executed by Windows machines in a WScript environment, JS code that is embedded into PDF files, the Node packages that are used for server-side applications and JavaScript files that are loaded in the webpage.

Samples are collected from a variety of sources. These sources include Node Package Manager (NPM) for widely used Node packages, MalwareBazaar and VirusTotal for malicious samples, and open-source repositories on GitHub and Kaggle. Additionally, general web scraping is employed to gather JavaScript files embedded in live webpages. The sources and the sample count obtained from sources is given on appendix 3.

A key consideration during dataset generation is the inherent structural differences between these JS files. WScript files are standalone and typically do not require external dependencies or further transformation. In contrast, JavaScript embedded in PDF files, Node packages, and web-based JavaScript often rely on external dependencies.

To address these challenges, a transformation process is implemented for non-standalone JavaScript samples. This process varies depending on the source:

1. **JavaScript in PDF Files:** PDF analysis tools are used to extract the script components, which are then isolated and analyzed as a standalone JS file.

2. **Node Packages:** Node packages frequently include dependencies. For analysis, the package is unpacked, and the dependency tree is flattened to ensure that all required JS code is available for analysis.
3. **JavaScript in Webpages:** Websites often load multiple JavaScript files asynchronously or through bundling. Web scraping tools are used to download entire web pages, and embedded scripts are extracted. This process helps ensure that dynamically injected scripts are not overlooked.

The details of the transformation process for each sample type are given on the following section 5.1.2.

5.1.2 Dataset transformation

The JS code in the PDF files is extracted by enumerating the PDF tree for all the objects containing **/JavaScript** where all the enumerated JS code are assembled as a single JS file. A sample of a parsed object from a PDF file is given in the figure 8, where the JS code is residing in the parenthesis following **/JS**.

```
1 0 obj
<<
  /Type /Catalog
  /Pages 2 0 R
  /OpenAction <<
   /S /JavaScript
   /JS (this.exportDataObject({ cName: "calc.exe",nLaunch: 2 });)
  >>
>>
endobj
```

Figure 8. PDF object containing JS code

For the Node packages, the module bundling is used [39]. Bundling is a technique where different JS files are combined into a single file, where the required functions and variables

are placed in the correct order in the final JS file. Bundlers can also include the code of the required dependencies in the final JS file, which can come from different NPM packages. This option is used in this research to ensure any external dependency exist on hybrid-analysis environment. The diagram of this process is given on the figure 10.

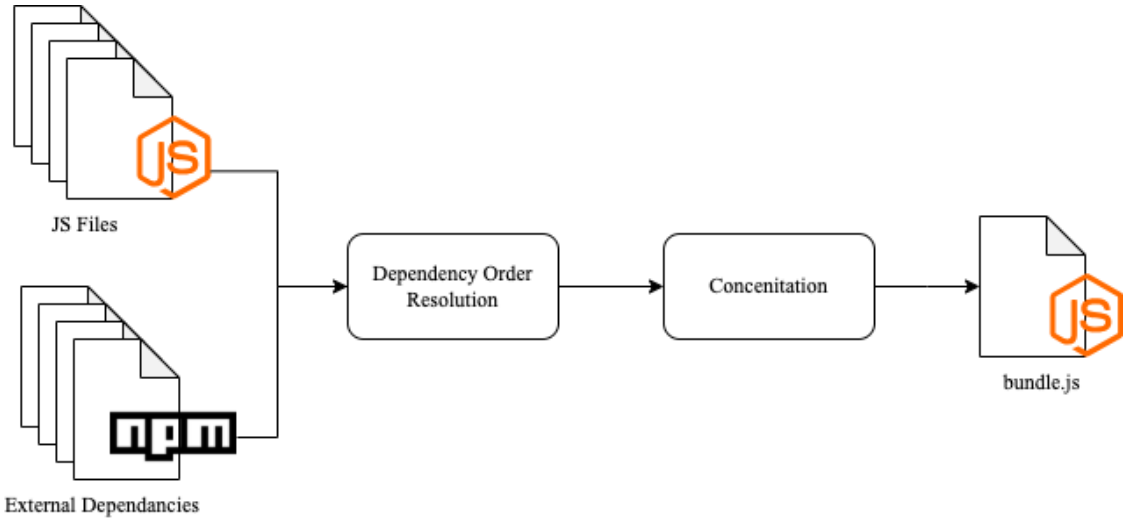

Figure 9. Diagram of the bundling process

Websites can load various JS files to function, which can also load additional JS files that are executed in the same context. This context can also be shared between other websites loaded into the current website, which can also load its own set of JS files. Additional to the JS files, the HTML elements might be required by the JS files used by the website. To generate the dataset for website samples, the hierarchy of the frames and loaded JS files by those frames are extracted by using a web browser. The HTML content of each frame and which frame the JS file is loaded is also included in the dataset for the website to be used on dynamic analysis. The diagram of this process is given on the figure 10.

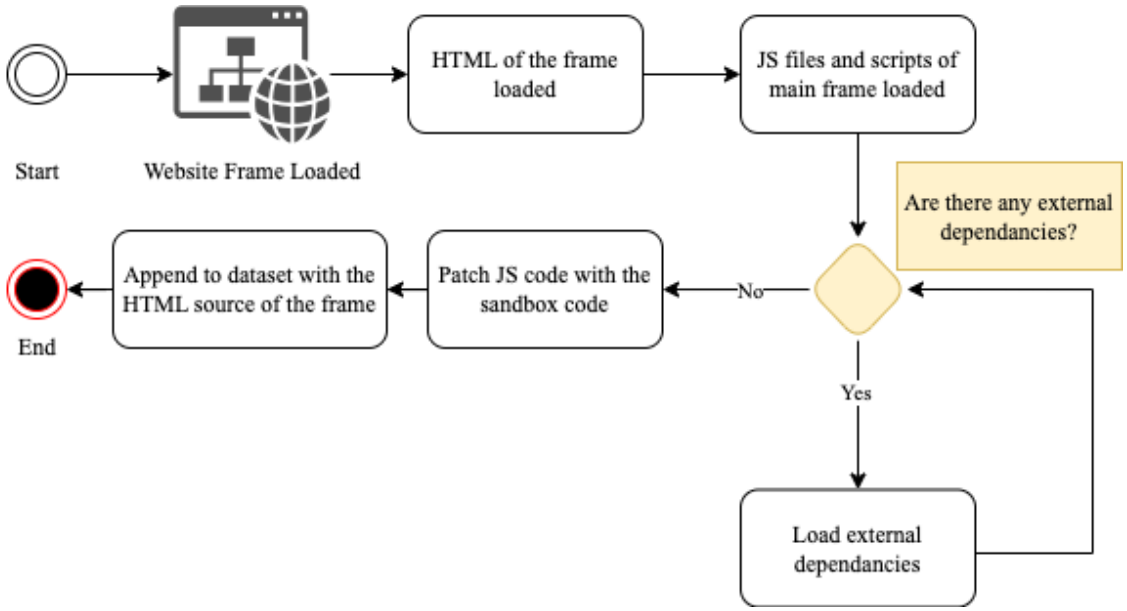

Figure 10. Diagram of website sample transformation process

5.1.3 Dataset labeling

Since the goal of this research is to classify the JavaScript files with the chosen classifications on the section 3; malware, ad/telemetry, skimmer and normal, the dataset needs to be labeled to compare against the model classification for evaluation.

The samples which are collected from the collections that match the criteria of the chosen classifications are labeled with their respective classifications. The remaining samples in the dataset that are collected by scraping or from a mixed collection are labeled manually according to the criteria.

5.1.4 Final Dataset

After the collection, transformation and labeling steps, the final dataset with the following structure is obtained.

Key	Purpose
`fileName`	Name of the file.
`fileContents`	JS source code of the file.
`fileAST`	AST of the source code.
`mainFrame`	Frame of the file, used by web samples.
`target`	Target of the file, can be web/pdf/package/wscript.
`sandboxOutput`	The dynamic-analysis sandbox output of the file.
`knownType`	The classification type which this sample labeled with.
`classifiedTypeA`	The classification result for using model A.
`classifiedTypeB`	The classification result for using model B.
`classifiedTypeC`	The classification result for using model C.

Table 2. Dataset fields and their purposes

The total size of the final dataset is 2685 with the distribution of labels given on figure 11.

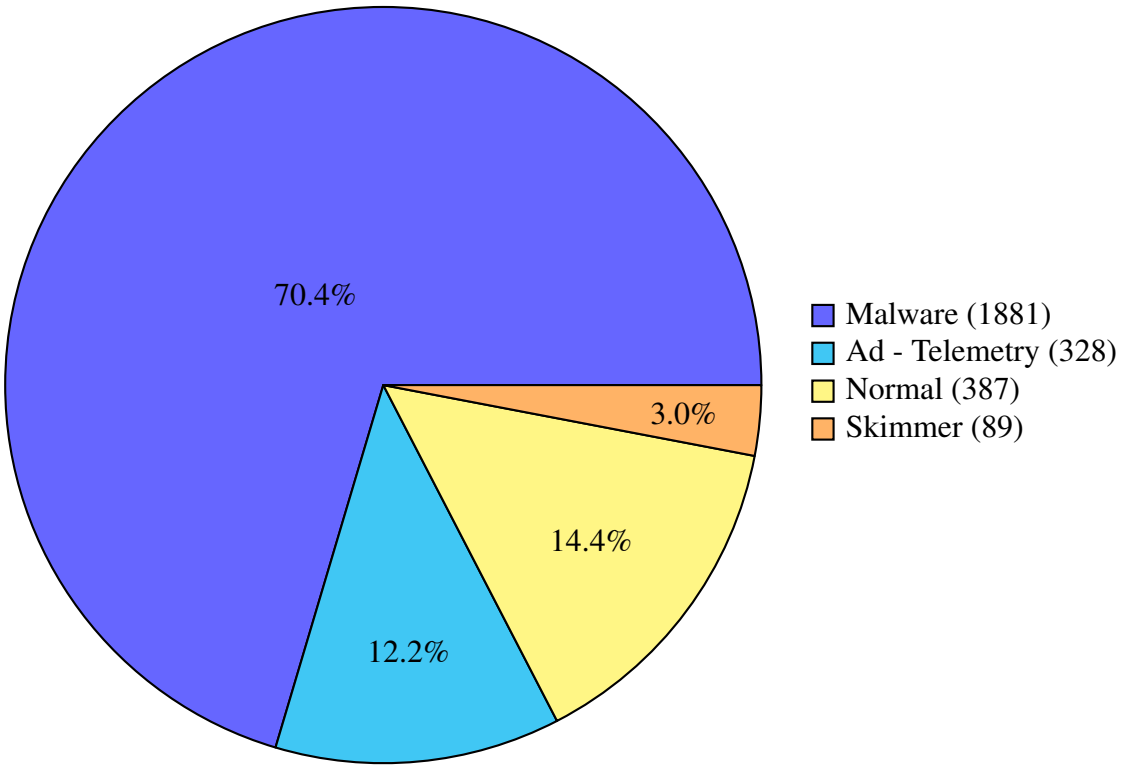

Figure 11. Distribution of the sample labels in dataset

5.2 Evaluation Setup

This section provides the details of how the input for large language models is derived, which metrics are used for evaluation, and the details of the models that will be used for evaluation.

To address context length constraints and potential financial limitations in reproducing this study, commercial models such as ChatGPT [40] or Claude [41] are not used in this research. Instead, the evaluation utilizes publicly available and locally runnable models given in the following section.

5.2.1 Model selection

The three models chosen for evaluation are Gemma2 with 9 billion parameters[42], Mistral with 7 billion parameters [43], and LLama2 with 7 billion parameters [44] with no additional fine-tuning applied to the models.

Throughout this thesis, the Gemma2 model will be referred to as **Model A**, the LLama2 model will be referred to as **Model B**, and the Mistral model will be referred to as **Model C**.

5.2.2 Evaluation Metrics

The metrics used for model evaluation are accuracy, precision, and recall where precision and recall will be evaluated per classification type to determine the performance of the LLMs for classification of the given type. The formula for each metric is given in the following figures.

$$\text{Accuracy} = \frac{\text{Successful classification}}{\text{Total classification}}$$

Figure 12. Accuracy Formula

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

Figure 13. Precision Formula

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

Figure 14. Recall Formula

5.2.3 Few-shot prompting

Few-shot prompting is a prompting technique used with LLMs where a model is provided with a small number of examples (few shots) within the input prompt to guide it in generating responses or performing tasks. [45]

The source code, AST, and sandbox output of a sample from each classification type is given as an example in the model before the classification inputs.

```
<<< A random sample from each type is appended to the prompt
    Sandbox.txt
    ```
    ${sandbox output}
    ```
    AST
    ```
    ${AST}
    ```
    sample.js
    ```
    ${source code}
    ```
    classification: malware / skimmer / ad/telemetry / normal
<<<
```

```
Sandbox.txt
```
${sandbox output}
```
AST
```
${AST}
```
sample.js
```
${source code}
```
Classify this JavaScript file as "normal", "malware","skimmer",
"ad/telemetry".
classification:
```

Figure 15. Few-shot prompt template to be used as input for the models

5.2.4 Zero-shot prompting

Zero-shot prompting is a prompting technique used with LLMs where a model is given a task without any specific training or examples for that task, making the model rely solely on its training data. [46] The Same prompting template as the few-shot prompting is used without the examples.

```
Sandbox.txt
```
${sandbox output}
```
AST
```
${AST}
```
sample.js
```
${source code}
```
Classify this JavaScript file as "normal", "malware","skimmer",
"ad/telemetry".
classification:
```

Figure 16. Zero-shot prompt template to be used as input for the models

6. Evaluation Result

6.1 Few-shot prompting results

This section summarizes the performance of each model when evaluated with few-shot prompting for the classification task. The results highlight the success and failure rates for different classification categories.

6.1.1 Model A

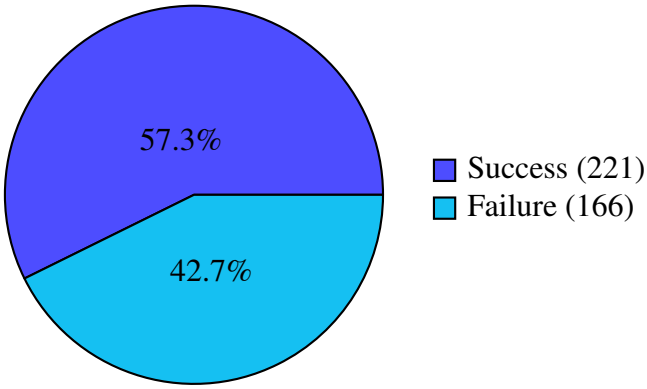

Figure 17. Success rate for "normal" using model A with few-shot prompting

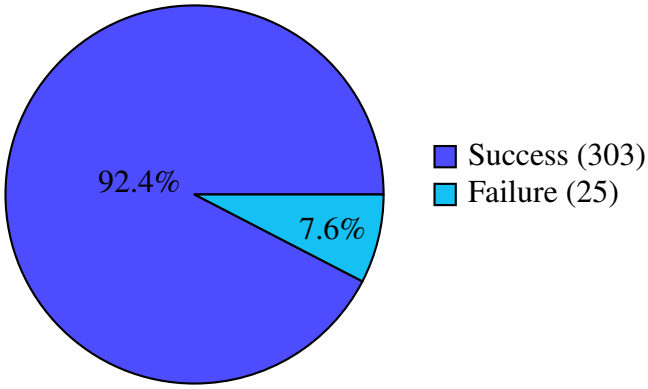

Figure 18. Success rate for "ad/telemetry" using model A with few-shot prompting

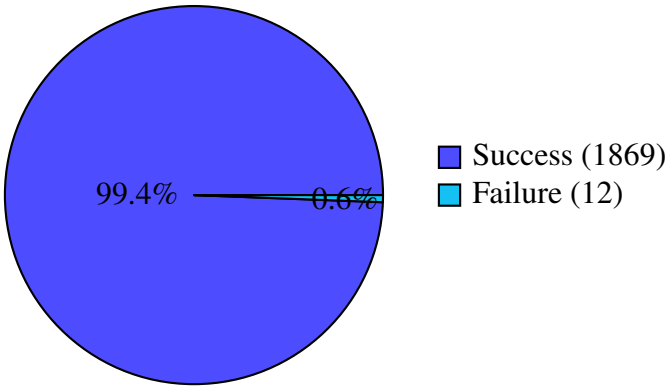

Figure 19. Success rate for "malware" using model A with few-shot prompting

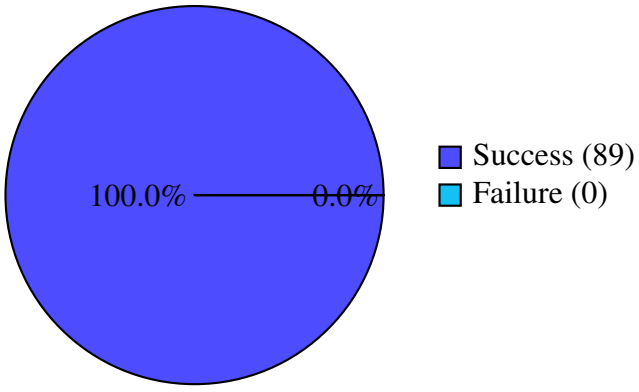

Figure 20. Success rate for "skimmer" using model A with few-shot prompting

6.1.2 Model B

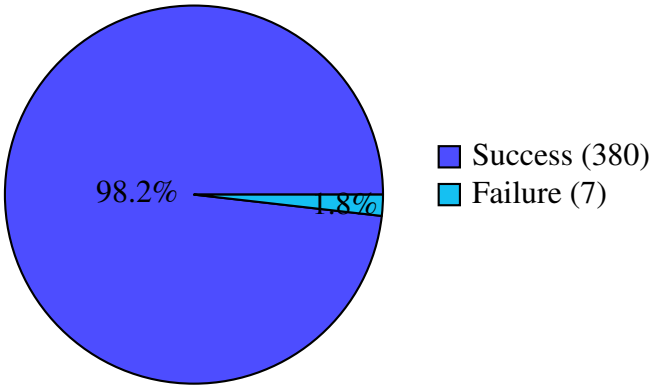

Figure 21. Success rate for "normal" using model B with few-shot prompting

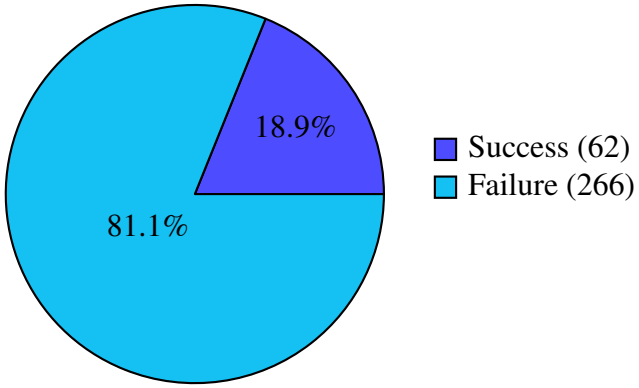

Figure 22. Success rate for "ad/telemetry" using model B with few-shot prompting

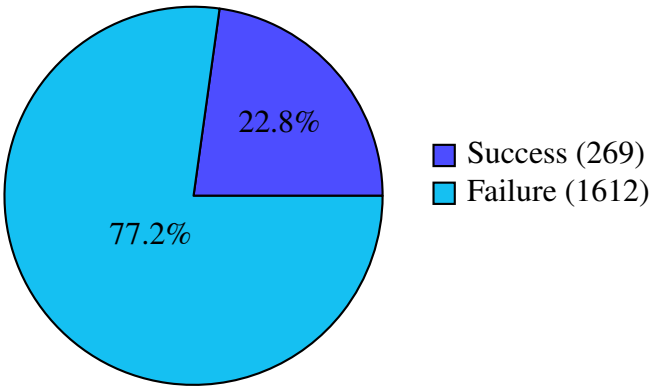

Figure 23. Success rate for "malware" using model B with few-shot prompting

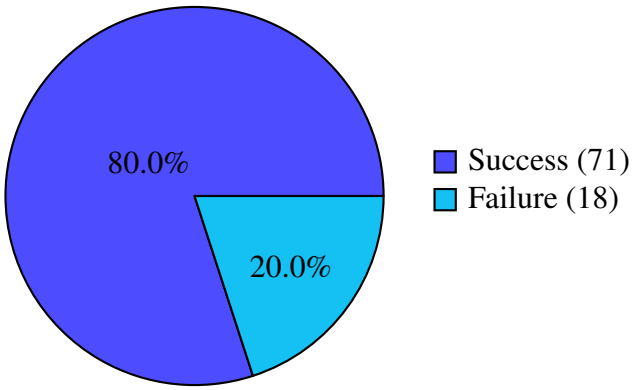

Figure 24. Success rate for "skimmer" using model B with few-shot prompting

6.1.3 Model C

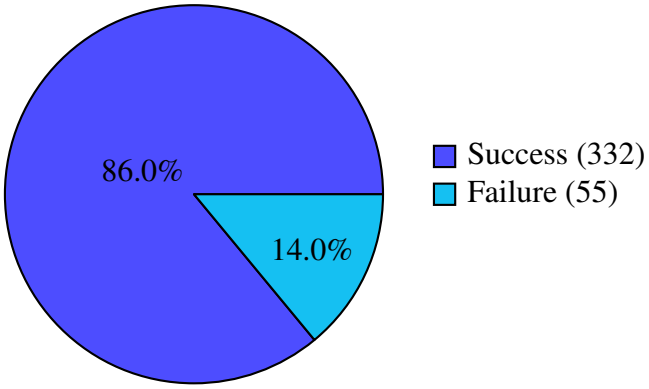

Figure 25. Success rate for "normal" using model C with few-shot prompting

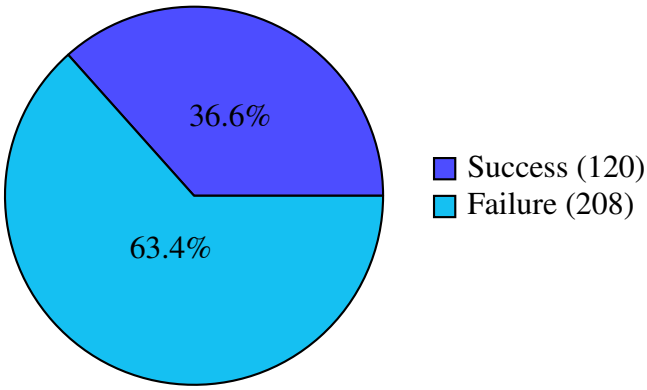

Figure 26. Success rate for "ad/telemetry" using model C with few-shot prompting

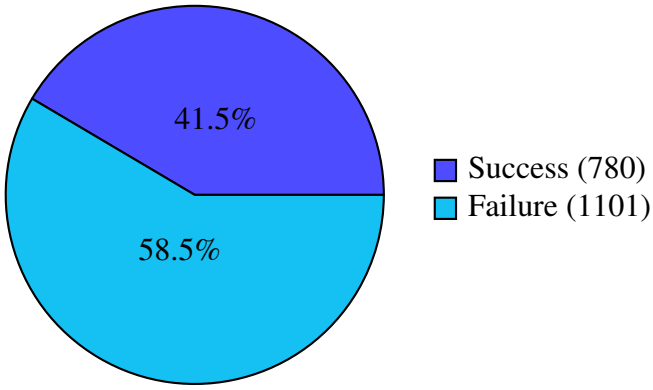

Figure 27. Success rate for "malware" using model C with few-shot prompting

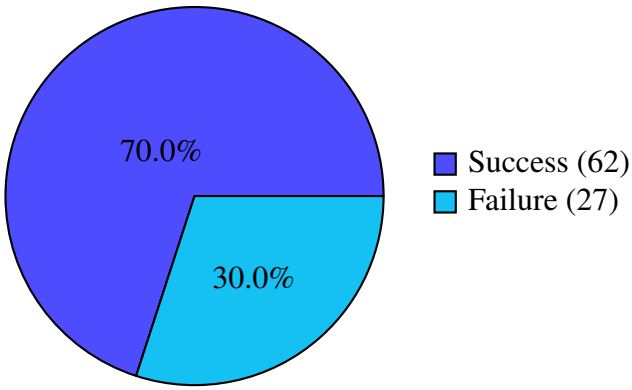

Figure 28. Success rate for "skimmer" using model C with few-shot prompting

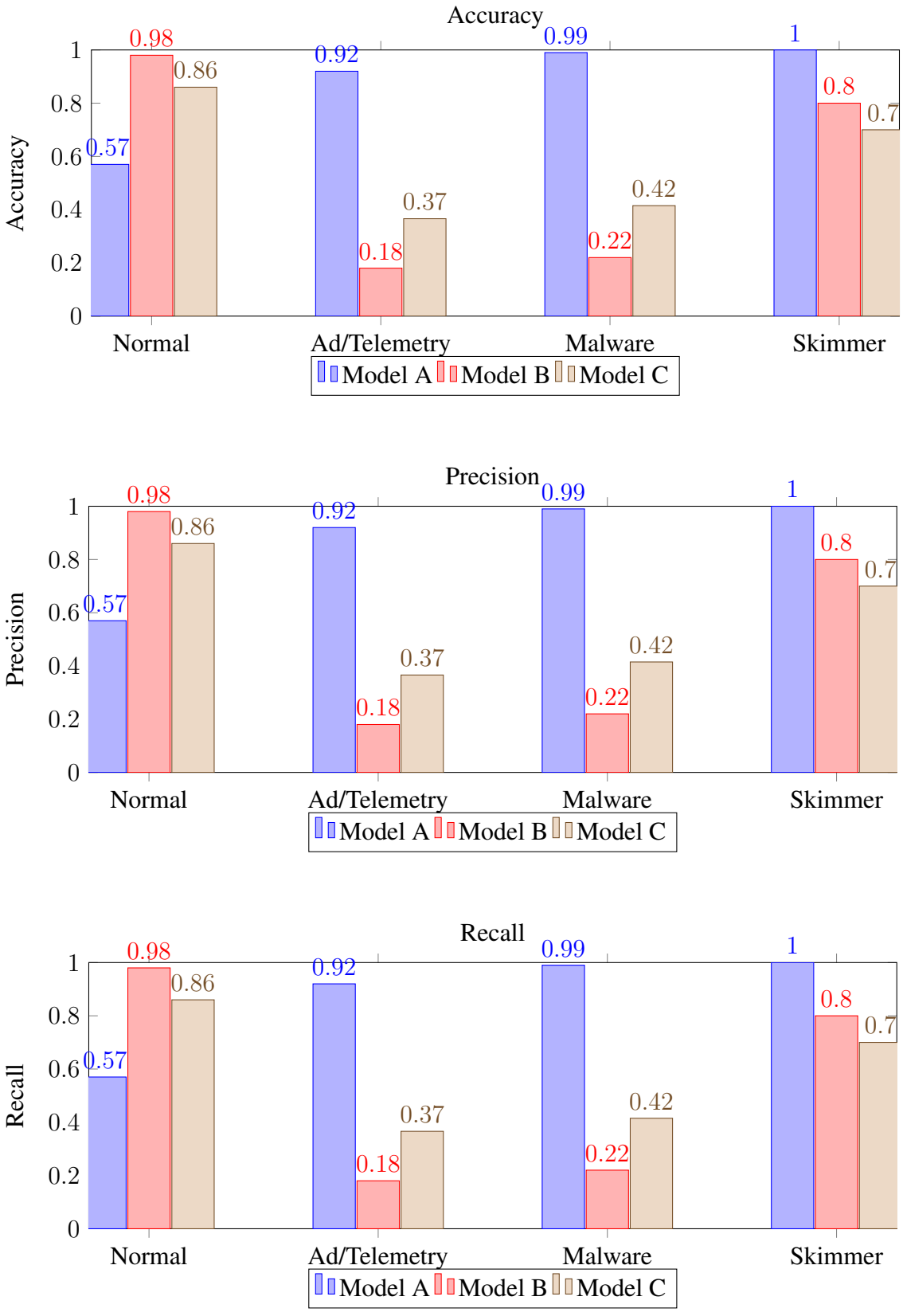

Figure 29. Accuracy, Precision, and Recall for Models A, B, and C by classification

6.2 Zero-shot prompting results

This section summarizes the performance of each model when evaluated with zero-shot prompting for the classification task. The results highlight the success and failure rates for different classification categories.

6.2.1 Model A

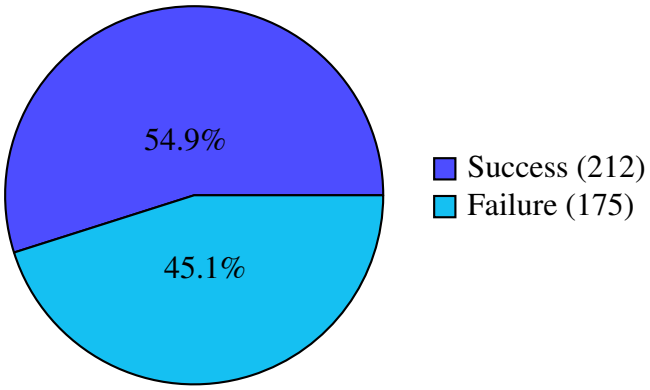

Figure 30. Success rate for "normal" using model A with zero-shot prompting

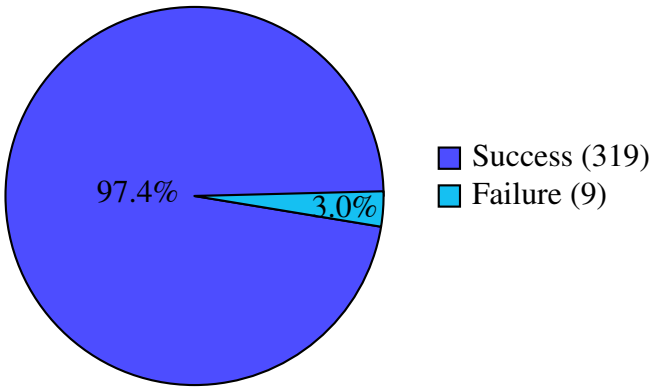

Figure 31. Success rate for "ad/telemetry" using model A with zero-shot prompting

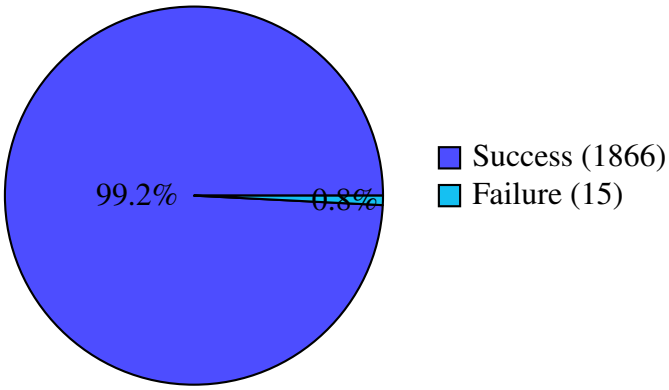

Figure 32. Success rate for "malware" using model A with zero-shot prompting

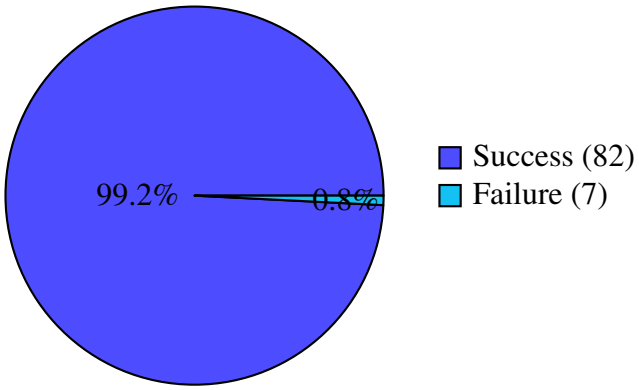

Figure 33. Success rate for "skimmer" using model A with zero-shot prompting

6.2.2 Model B

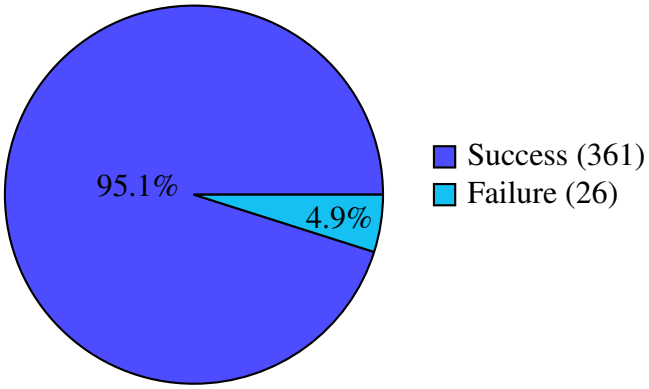

Figure 34. Success rate for "normal" using model B with zero-shot prompting

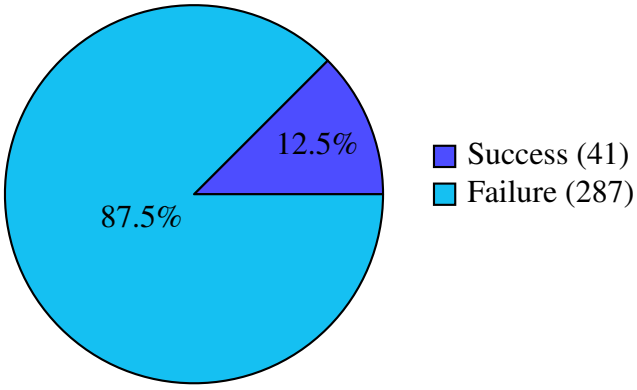

Figure 35. Success rate for "ad/telemetry" using model B with zero-shot prompting

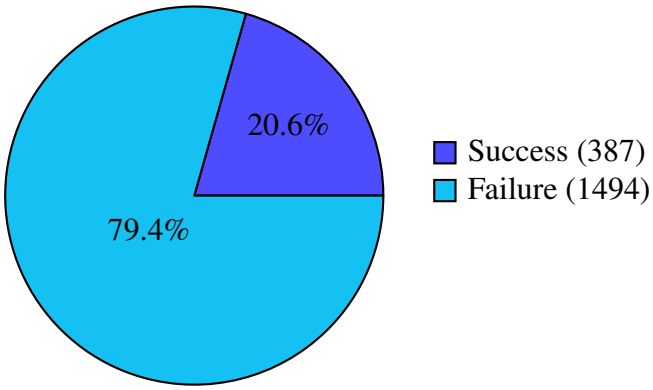

Figure 36. Success rate for "malware" using model B with zero-shot prompting

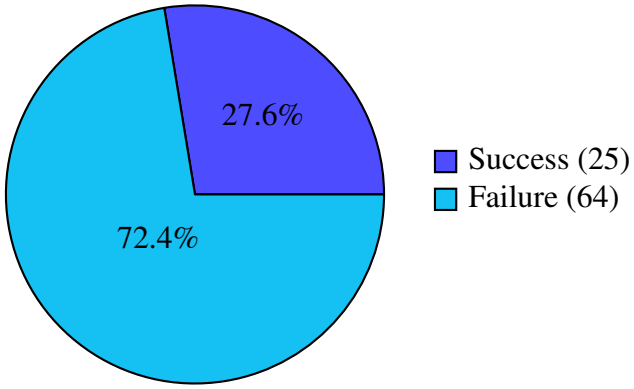

Figure 37. Success rate for "skimmer" using model B with zero-shot prompting

6.2.3 Model C

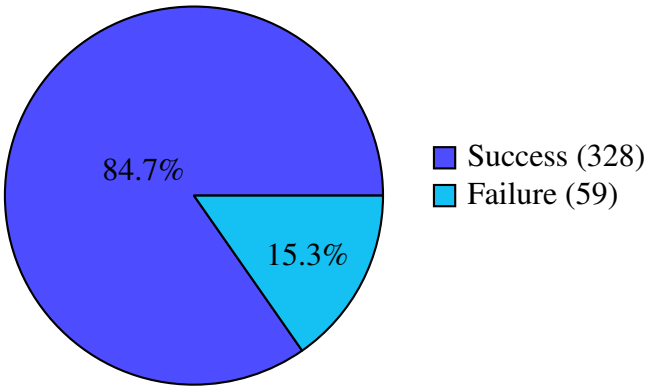

Figure 38. Success rate for "normal" using model C with zero-shot prompting

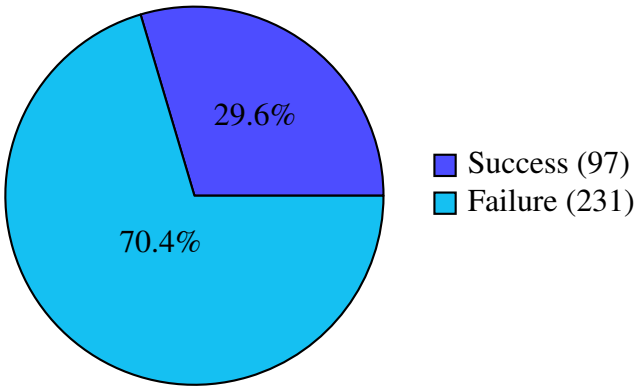

Figure 39. Success rate for "ad/telemetry" using model C with zero-shot prompting

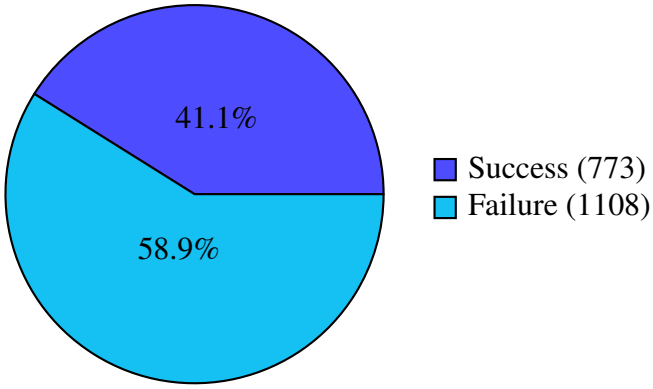

Figure 40. Success rate for "malware" using model C with zero-shot prompting

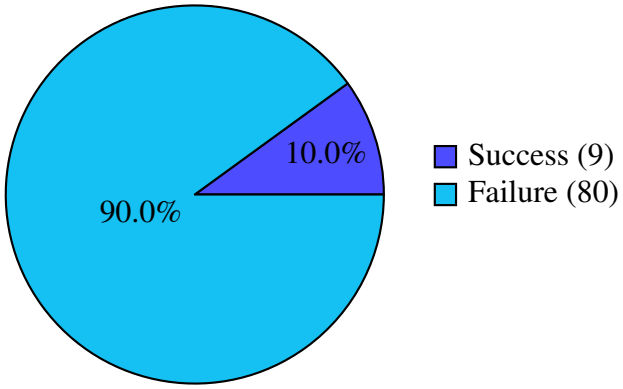

Figure 41. Success rate for "skimmer" using model C with zero-shot prompting

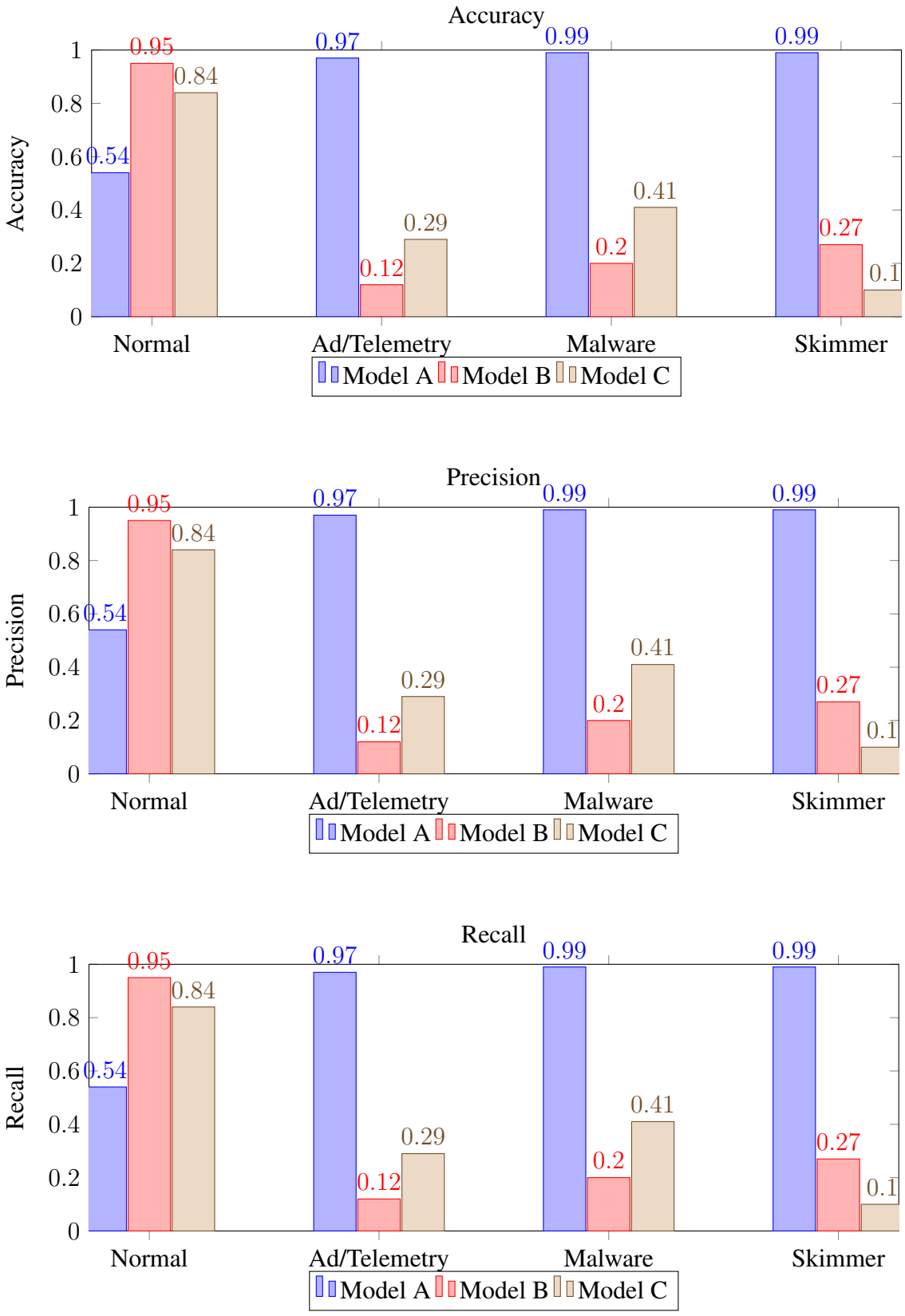

Figure 42. Accuracy, Precision, and Recall for Models A, B, and C by classification

6.3 Analysis of the results

On the zero-shot prompting, model A demonstrated success in the specialized classifications, achieving an accuracy of 97.4% for classifying files that are ad/telemetry, 99.2% for classifying the files that are malware and skimmer but faltered in classification of normal files with the success rate of 54.9%.

In few-shot prompting, model A continued to demonstrate success in the specialized classifications, achieving 92.4% accuracy for ad/telemetry, 99.4% for Malware, and 100% for skimmer classification. With a slight but not significant increase in accuracy at the classification of normal files with a success rate of 57.3%.

It imbalance of accuracy of normal classification against specialized classifications suggests that specialized classifications have distinct patterns or features in their input, whether on the AST, source code, or sandbox result, that the model readily identifies even in a zero-shot context. The lower accuracy on normal files can be either due to a lack of features or patterns in the input that can stand out to specify the given sample as a normal JS file or the overall features and patterns overlapping with the other categories that were in the training of the model. The slight improvement in the accuracy in few-shot prompting suggests that the problem might be the overlapping issue because the additional examples of each type helped the model's understanding of the normal type although minimal.

Using zero-shot prompting, model B demonstrated strong performance in the classification of normal files with a success rate of 95.1% but struggled significantly in specialized classifications with the accuracy on classifying ad/telemetry files being 18.9%, malware files being 22.7% and skimmer files being 80%. In few-shot prompting, model B had an increased accuracy for classifying normal files with a 98.2% success rate, however, its accuracy on specialized classification remained low, with success rates of 18.9% for ad/telemetry, 22.8% for malware and 80% for skimmer files.

The imbalance in the performance across classifications suggests that model B is a model that is more generalized and effective for common patterns but lacks the training or fine-tuning for identifying the specialized files.

On zero-shot prompting, model C showed balanced but moderate results across all classifications with an accuracy of 29.6% for ad/telemetry classification, 41.1% for malware classification, and 10% for skimmer classification while performing relatively better for normal file classification with an accuracy of 84.7%. In few-shot prompting, model C displayed an improvement in classification with an accuracy of 41.5% for malware files,

36.6% for ad/telemetry files, and 70% for normal files. The relatively higher accuracy in classifying normal files suggests that model C is also great at classifying typical files but struggles to classify specialized files which can be due to the training of the model C being conducted on a more generalized dataset. Its performance on specialized files highlights its need for further training or fine-tuning with the samples from specialized specifications.

7. Discussion

This section will serve as a discussion on overall analysis results and try to answer the research questions defined after the literature review based on which were defined at the beginning of the research:

- **RQ1:** Is it possible to use LLMs with hybrid analysis for classification accurately?
- **RQ2:** When using LLMs for classification, are there any accuracy changes when few-shot prompting is used?
- **RQ3:** Will different LLMs result in the same classifications with the same input?

Is it possible to use LLMs with hybrid analysis for classification accurately?

The analysis result shows that the LLMs can be used with hybrid analysis for classification, but their accuracy heavily depends on the specific model, the type of data, and the classification categories. The three models used in the analysis resulted in different performances across classification categories with major differences.

Model A showed its strength in specialized classification with high accuracy even in zero-shot prompting with no examples but struggled with the identification of normal files with a high false positive rate indicating that it is training data or architecture might not focus on less distinctive patterns of normal JS files.

Model B demonstrated high accuracy in classifying normal files but poor accuracy in specialized classifications. This suggests that model B is optimized for general patterns while lacking the fine-tuning required for specialized classification.

Model C showed moderate but weak results across all classification categories showing its generalist training which suggests without additional training, it is not feasible to use it for this task.

In summary, the answer to the research question is: **It is possible to use LLMs with hybrid analysis for classification; however, their accuracy depends on their pre-training. Fine-tuning can be used to increase accuracy and achieve better results for this task.**

When using LLMs for classification, are there any accuracy changes when few-shot prompting is used?

Based on the analysis results, few-shot prompting showed a general improvement in classification accuracy across different LLMs. When few-shot prompting results compared to zero-shot prompting results, only a minor improvement in file classifications is observed at an average of 3%. This indicates that additional examples in the few-shot prompts help the model to disambiguate the overlapping features but for a slight accuracy gain.

Overall, the question can be answered with **few-shot prompting can improve accuracy by helping models to understand the context better to some extent, but the underlying model training and fine-tuning remain the critical component for the classification task**.

Will different LLMs result in the same classifications with the same input?

When provided with the same input, Model A excelled in specialized classifications but struggled with generalized ones, indicating its strength in identifying distinct patterns but difficulty in handling overlapping features. Model B performed well in generalized classifications, but it was falling short in specialized tasks, suggesting a broader but less targeted understanding of it. Model C maintained a balanced yet lower overall accuracy, reflecting its more generalized training approach. Based on these results the question can be answered that **different LLMs yield varying classifications when provided with the same input, reflecting differences in their training methodologies, architectures, and domain specializations for the classification task**.

8. Conclusion

In this research, a systematic literature review is conducted based on the topic and theoretical background, and the gap in the literature related to hybrid analysis and LLMs is identified. The methodology for testing and validation is defined and a dataset is generated to further conduct the empirical testing with the development and implementation details of the testing environment explained.

The tests are conducted with the selected models with the generated dataset and results are analyzed individually and as a whole, the research questions are answered based on the evaluation results followed by the limitations of this research and future work on this topic. The findings show the promising success of the proposed approach to JS file detection in the topic and underscore the importance of model selection, prompting techniques, and dataset quality in achieving robust results. Future work could involve refining hybrid analysis methodologies and expanding model capabilities to address the limitations observed, ultimately contributing to better cyberthreat detection.

8.1 Limitations

During the research on this study, several limitations were encountered that impacted its scope and findings.

The primary limitation was the constraints on usable samples from the dataset. While the initial dataset comprised 16,000 JS files, only 2638 were included in the final dataset with the criteria of being successfully patched on the AST level for dynamic analysis. This reduced dataset hindered the overall sample size of the research during evaluation.

Another limitation was the limitations on model selection and fine-tuning due to constraints on computational resources. The evaluation was carried out with limited computational resources that restricted the use of more capable LLMs for evaluation. The selected models, Gemma2 (9B), Llama2 (7B), and Mistral (7B), were evaluated without additional fine-tuning. While this approach reflects real-world applications where pre-trained models are deployed, it constrained the ability to fully evaluate the LLMs for this specific task, potentially impacting classification accuracy, particularly in specialized categories.

8.2 Future Work

To overcome these limitations given above, future work can use a different dynamic analysis approach where the JS samples are instrumented not through the code by AST patching but from the runtime by having a custom JavaScript execution engine that logs the similar output that is used in this research during execution in the native code, without affecting the code. This approach can allow the full initial dataset to be used for evaluation.

Another future work is to use the labeled dataset to fine-tune an LLM that is specialized for the classification task to further show the strength of the hybrid-analysis approach. Fine-tuning a model with the dataset could improve accuracy, particularly in specialized classifications, by adapting the model to patterns in those files as well as reducing the false positives in normal file classifications.

Future work can also explore the integration of more feature representations, such as incorporating runtime behavior graphs or enhanced metadata from the JavaScript execution environment such as the website's HTML source code. These additions might improve the models' ability to differentiate between closely related classifications and adapt to real-world use cases.

References

[1] Allen Wirfs-Brock and Brendan Eich. “JavaScript: the first 20 years”. In: *Proceedings of the ACM on Programming Languages* 4.HOPL (June 2020), pp. 1–189. ISSN: 2475-1421. DOI: 10.1145/3386327.

[2] Kristoffer Gunnarsson and Olivia Herber. *The most popular programming languages of GitHub's trending repositories*. 2020.

[3] Peter G. Aitken. *Windows script host. [scripting solutions for Windows 2000, Windows 9x, and Windows ME; covers both VBSript and JSript; step-by-step coverage of creating, running, scheduling, and distrituting scripts; system administration, application control, file management, messaging, database access, and more; creating high-performance WSH script]*. Prentice Hall PTR Microsoft Technologies Series. Upper Saddle River, N.J.[u.a.]: Prentice Hall, 2001. 364 pp. ISBN: 0130287016.

[4] Soojin Yoon, JongHun Jung, and HwanKuk Kim. “Attacks on Web browsers with HTML5”. In: *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*. IEEE, Dec. 2015, pp. 193–197. DOI: 10.1109/icitst.2015.7412087.

[5] Farzad Nourmohammadzadeh Motlagh et al. *Large Language Models in Cybersecurity: State-of-the-Art*. 2024. DOI: 10.48550/ARXIV.2402.00891.

[6] Aurore Fass et al. “JaSt: Fully Syntactic Detection of Malicious (Obfuscated) JavaScript”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer International Publishing, 2018, pp. 303–325. ISBN: 9783319934112. DOI: 10.1007/978-3-319-93411-2_14.

[7] Omer Tripp and Omri Weisman. “Hybrid analysis for javascript security assessment”. In: *ESEC/FSE*. Vol. 11. Citeseer. 2011.

[8] Amod Narendra Narvekar and Kiran K. Joshi. “Security sandbox model for modern web environment”. In: *2017 International Conference on Nascent Technologies in Engineering (ICNTE)*. IEEE, Jan. 2017, pp. 1–6. DOI: 10.1109/icnte.2017.7947885.

[9] Paul Jensen. *Cross-Platform Desktop Applications. Using Node, Electron, and NW.js*. Description based on publisher supplied metadata and other sources. New York: Manning Publications Co. LLC, 2017. 1273 pp. ISBN: 9781638353928.

[10] Brandon Satrom. *Building polyfills. Web platform APIs for the present and future*. 1. ed. Beijing: O'Reilly, 2014. 152 pp. ISBN: 9781449370732.

[11] Fuad Budagov et al. “A Systematic Literature Review on Applicability of Robot Assistants in Higher Education”. In: *Methodologies and Intelligent Systems for Technology Enhanced Learning, 14th International Conference*. Ed. by Christothea Herodotou et al. Cham: Springer Nature Switzerland, 2024, pp. 21–32. ISBN: 978-3-031-73538-7. DOI: 10.1007/978-3-031-73538-7_3.

[12] K. Jasmine et al. “Malware Detection and Classification with Deep Learning Models”. In: *2023 International Conference on Applied Intelligence and Sustainable Computing (ICAISC)* (2023). DOI: 10.1109/ICAISC58445.2023.10199581.

[13] *Detecting malicious JavaScript code based on semantic analysis* (2020).

[14] *JStrong: Malicious JavaScript detection based on code semantic representation and graph neural network* (2022).

[15] Yunhua Huang et al. “JSContana: Malicious JavaScript detection using adaptable context analysis and key feature extraction”. In: *Computers amp; Security* 104 (May 2021), p. 102218. ISSN: 0167-4048. DOI: 10.1016/j.cose.2021.102218.

[16] Tiago Brito et al. “Study of JavaScript Static Analysis Tools for Vulnerability Detection in Node.js Packages”. In: *IEEE Transactions on Reliability* (2023). DOI: 10.1109/tr.2023.3286301.

[17] Alvin T. S. Chan et al. “Transformer-based Vulnerability Detection in Code at EditTime: Zero-shot, Few-shot, or Fine-tuning?” In: *arXiv.org* (2023). DOI: 10.48550/arxiv.2306.01754.

[18] Bodin Chinthanet et al. “Code-based Vulnerability Detection in Node.js Applications: How far are we?” In: *International Conference on Automated Software Engineering* (2020). DOI: 10.1145/3324884.3421838.

[19] Moumita Das Purba et al. “Software Vulnerability Detection using Large Language Models”. In: *2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW)* (2023). DOI: 10.1109/issrew60843.2023.00058.

[20] Zeyu Gao et al. “How Far Have We Gone in Vulnerability Detection Using Large Language Models”. In: *arXiv.org* (2023). DOI: 10.48550/arxiv.2311.12420.

[21] K. J. Ren et al. “An Empirical Study on the Effects of Obfuscation on Static Machine Learning-Based Malicious JavaScript Detectors”. In: *null* (2023). DOI: 10.1145/3597926.3598146.

[22] *TransAST: A Machine Translation-Based Approach for Obfuscated Malicious JavaScript Detection* (2023).

[23] *Statically Detecting JavaScript Obfuscation and Minification Techniques in the Wild* (2021).

[24] *Malicious Code Utilization Chain Detection Scheme based on Abstract Syntax Tree* (2022).

[25] Ammar Alazab et al. "Detection of Obfuscated Malicious JavaScript Code". In: *Future Internet* (2022). DOI: 10.3390/fi14080217.

[26] Muhammad Fakhrur Rozi et al. "Deep Neural Networks for Malicious JavaScript Detection Using Bytecode Sequences". In: *null* (2020). DOI: 10.1109/ijcnn48605.2020.9207134.

[27] *Malware Detection with LSTM using Opcode Language* (2019).

[28] P. V. V. Kishore et al. "JavaScript malware behaviour analysis and detection using sandbox assisted ensemble model". In: *IEEE Region 10 Conference* (2020). DOI: 10.1109/tencon50793.2020.9293847.

[29] Xincheng He et al. "Malicious JavaScript Code Detection Based on Hybrid Analysis". In: *null* (2018). DOI: 10.1109/apsec.2018.00051.

[30] Masudul Hasan Masud Bhuiyan et al. "SecBench.js: An Executable Security Benchmark Suite for Server-Side JavaScript". In: *International Conference on Software Engineering* (2023). DOI: 10.1109/icse48619.2023.00096.

[31] Takashi Koide, Hiroki Nakano, and Daiki Chiba. "ChatPhishDetector: Detecting Phishing Sites Using Large Language Models". In: *IEEE Access* (2024). DOI: 10.1109/access.2024.3483905.

[32] Muhammad Fakhrur Rozi et al. "Understanding the Influence of AST-JS for Improving Malicious Webpage Detection". In: *Applied Sciences* (2022). DOI: 10.3390/app122412916.

[33] Xincheng He, Lei Xu, and Chunliu Cha. "Malicious JavaScript Code Detection Based on Hybrid Analysis". In: *2018 25TH ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE (APSEC 2018)*. Asia-Pacific Software Engineering Conference. 25th Asia-Pacific Software Engineering Conference (APSEC), Nara, JAPAN, DEC 04-07, 2018. Special Interest Grp Software Engn, Informat Proc Soc Japan. 2018, pp. 365–374. ISBN: 978-1-7281-1970-0. DOI: 10.1109/APSEC.2018.00051.

[34] Jordan Jueckstock and Alexandros Kapravelos. "VisibleV8: In-browser Monitoring of JavaScript in the Wild". In: *IMC'19: PROCEEDINGS OF THE 2019 ACM INTERNET MEASUREMENT CONFERENCE*. ACM Internet Measurement Conference (IMC), Univ Twente, Amsterdam, NETHERLANDS, OCT 21-23, 2019. Assoc Comp Machinery; ACMSIGCOMM; ACM SIGMETRICS; USENIX; Assoc

Comp Machinery. 2019, pp. 393–405. ISBN: 978-1-4503-6948-0. DOI: 10.1145/3355369.3355599.

[35] Antoine Lemay and Sylvain P. Leblanc. "Is eval () Evil : A study of JavaScript in PDF malware". In: *PROCEEDINGS OF THE 2018 13TH INTERNATIONAL CONFERENCE ON MALICIOUS AND UNWANTED SOFTWARE (MALWARE 2018)*. 13th International Conference on Malicious and Unwanted Software (MALWARE), MA, OCT 22-24, 2018. IEEE Comp Soc. 2018, pp. 13–22. ISBN: 978-1-7281-0155-2.

[36] Chandra Sekhar Biswal and Subhendu Kumar Pani. *Cyber-Crime Prevention Methodology*. Jan. 2021. DOI: 10.1002/9781119711629.ch14.

[37] Zhiyuan Wan et al. "Practical and effective sandboxing for Linux containers". In: *Empirical Software Engineering* 24 (2019), pp. 4034–4070.

[38] Thanh Bui. "Analysis of Docker Security". In: (Jan. 2015). DOI: 10.48550/ARXIV.1501.02967. arXiv: 1501.02967 [cs.CR].

[39] Mohamed Bouzid. *Webpack for Beginners: Your Step-by-Step Guide to Learning Webpack 4*. Apress, 2020. ISBN: 9781484258965. DOI: 10.1007/978-1-4842-5896-5.

[40] OpenAI. *ChatGPT*. https://chat.openai.com. 2024.

[41] Anthropic. *Claude: Anthropic's AI Assistant*. Accessed: 2024-12-22. 2024. URL: https://www.anthropic.com/claude.

[42] Gemma Team et al. *Gemma*. 2024. DOI: 10.34740/KAGGLE/M/3301.

[43] Albert Q. Jiang et al. "Mistral 7B". In: (Oct. 2023). DOI: 10.48550/ARXIV.2310.06825. arXiv: 2310.06825 [cs.CL].

[44] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. DOI: 10.48550/ARXIV.2307.09288.

[45] Archit Parnami and Minwoo Lee. "Learning from Few Examples: A Summary of Approaches to Few-Shot Learning". In: (Mar. 2022). DOI: 10.48550/ARXIV.2203.04291. arXiv: 2203.04291 [cs.LG].

[46] Farhad Pourpanah et al. "A Review of Generalized Zero-Shot Learning Methods". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (Nov. 2020), pp. 1–20. ISSN: 1939-3539. DOI: 10.1109/tpami.2022.3191696. arXiv: 2011.08641 [cs.CV].

Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis[1]

I Ali Atakan Basaran

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Evaluating large language models for JavaScript file classification using hybrid analysis", supervised by Fuad Budagov
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

02.01.2025

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 - AST representation of the sample JS code


```
{
  "type": "Program",
  "body": [
    {
      "type": "FunctionDeclaration",
      "id": {
        "type": "Identifier",
        "name": "answer",
        "range": [
          9,
          15
        ]
      },
      "params": [],
      "body": {
        "type": "BlockStatement",
        "body": [
          {
            "type": "VariableDeclaration",
            "declarations": [
              {
                "type": "VariableDeclarator",
                "id": {
                  "type": "Identifier",
                  "name": "x",
                  "range": [
                    88,
                    89
                  ]
                },
                "init": {
                  "type": "Literal",
                  "value": 42,
                  "raw": "42",
                  "range": [
                    92,
                    94
                  ]
                },
                "range": [
                  88,
                  94
                ]
              }
            ],
            "kind": "const",
            "range": [
              82,
              94
            ]
          },
          {
            "type": "ExpressionStatement",
            "expression": {
              "type": "CallExpression",
              "callee": {
                "type": "MemberExpression",
                "computed": false,
                "object": {
                  "type": "Identifier",
                  "name": "console",
                  "range": [
                    99,
                    106
                  ]
                },
                "property": {
                  "type": "Identifier",
                  "name": "log",
                  "range": [
                    107,
                    110
                  ]
```

```
                },
                "range": [
                  99,
                  110
                ]
              },
              "arguments": [
                {
                  "type": "Identifier",
                  "name": "x",
                  "range": [
                    111,
                    112
                  ]
                }
              ],
              "range": [
                99,
                113
              ]
            },
            "range": [
              99,
              114
            ]
          }
        ],
        "range": [
          18,
          116
        ]
      },
      "generator": false,
      "expression": false,
      "async": false,
      "range": [
        0,
        116
      ]
    },
    {
      "type": "ExpressionStatement",
      "expression": {
        "type": "CallExpression",
        "callee": {
          "type": "Identifier",
          "name": "answer",
          "range": [
            118,
            124
          ]
        },
        "arguments": [],
        "range": [
          118,
          126
        ]
      },
      "range": [
        118,
        127
      ]
    }
  ],
  "sourceType": "module",
  "range": [
    0,
    127
  ]
}
```

Appendix 3 - Dataset Sources

Source	Sample Count	Source	Sample Count
GitHub	1678	msn.com	3
Kaggle	50	qq.com	2
VirusTotal	215	office.com	3
MalwareBazaar	95	gap.com	12
tripadvisor.com	1	zillow.com	1
bbc.co.uk	5	ask.com	4
wikia.com	3	washingtonpost.com	8
naver.com	7	paypal.com	7
youth.cn	4	udemy.com	2
eksisozluk.com	3	target.com	2
sina.com.cn	1	tudou.com	13
baidu.com	2	chinadaily.com.cn	4
youku.com	14	capitalone.com	6
mozilla.org	10	yelp.com	2
mega.nz	1	walmart.com	4
fiverr.com	1	uol.com.br	18
www.npmjs.com	2	wix.com	7
sohu.com	6	hdfcbank.com	12
moodle.com	1	uptodown.com	1
pixnet.net	2	samsung.com	8
outbrain.com	5	bbc.com	4
popads.net	2	mediafire.com	5
chase.com	3	indeed.com	2
wellsfargo.com	8	aliexpress.com	22
globo.com	1	onedio.com	4
google.co.za	1	theguardian.com	11
alipay.com	2	alibaba.com	7
dailymail.co.uk	1	diply.com	5
slideshare.net	2	yahoo.com	5
orange.fr	5	instagram.com	1
salesforce.com	38	nytimes.com	15
steamcommunity.com	6	t-mobile.com	24

Source	Sample Count
soundcloud.com	9
foxnews.com	7
youtube.com	5
360.cn	5
taltech.ee	2
huffingtonpost.com	4
yandex.ru	1
mail.ru	2
milliyet.com.tr	12
stackexchange.com	1
ok.ru	5
cnet.com	2
pinterest.com	1
myway.com	3
tumblr.com	23
cnn.com	10
americanexpress.com	9
gmw.cn	2
wikihow.com	3
zhihu.com	4
aol.com	7
wikipedia.org	1
stackoverflow.com	4
9gag.com	2
daum.net	6
varzesh3.com	2
ameblo.jp	1
etsy.com	1
deviantart.com	3
vimeo.com	1
playstation.com	18
oracle.com	3
roblox.com	35
tmall.com	4
ebay.com	6

Source	Sample Count
linkedin.com	2
flipkart.com	1
indiatimes.com	8
vk.me	5
airbnb.com	3
menthorq.com	5
bet365.com	1
taobao.com	3
xfinity.com	3
imgur.com	5
bilibili.com	7
twitch.tv	20
flickr.com	4
ikea.com	7
booking.com	9
w3schools.com	6
forbes.com	3
amazon.com	4
fc2.com	1
jd.com	13
ibm.com	3
craigslist.org	1
microsoft.com	6
vice.com	8
rakuten.co.jp	24
buzzfeed.com	1
go.com	8
feedly.com	1
weather.com	9
apple.com	7
quora.com	4
so.com	3
spotify.com	3
malwarebytes.com	11