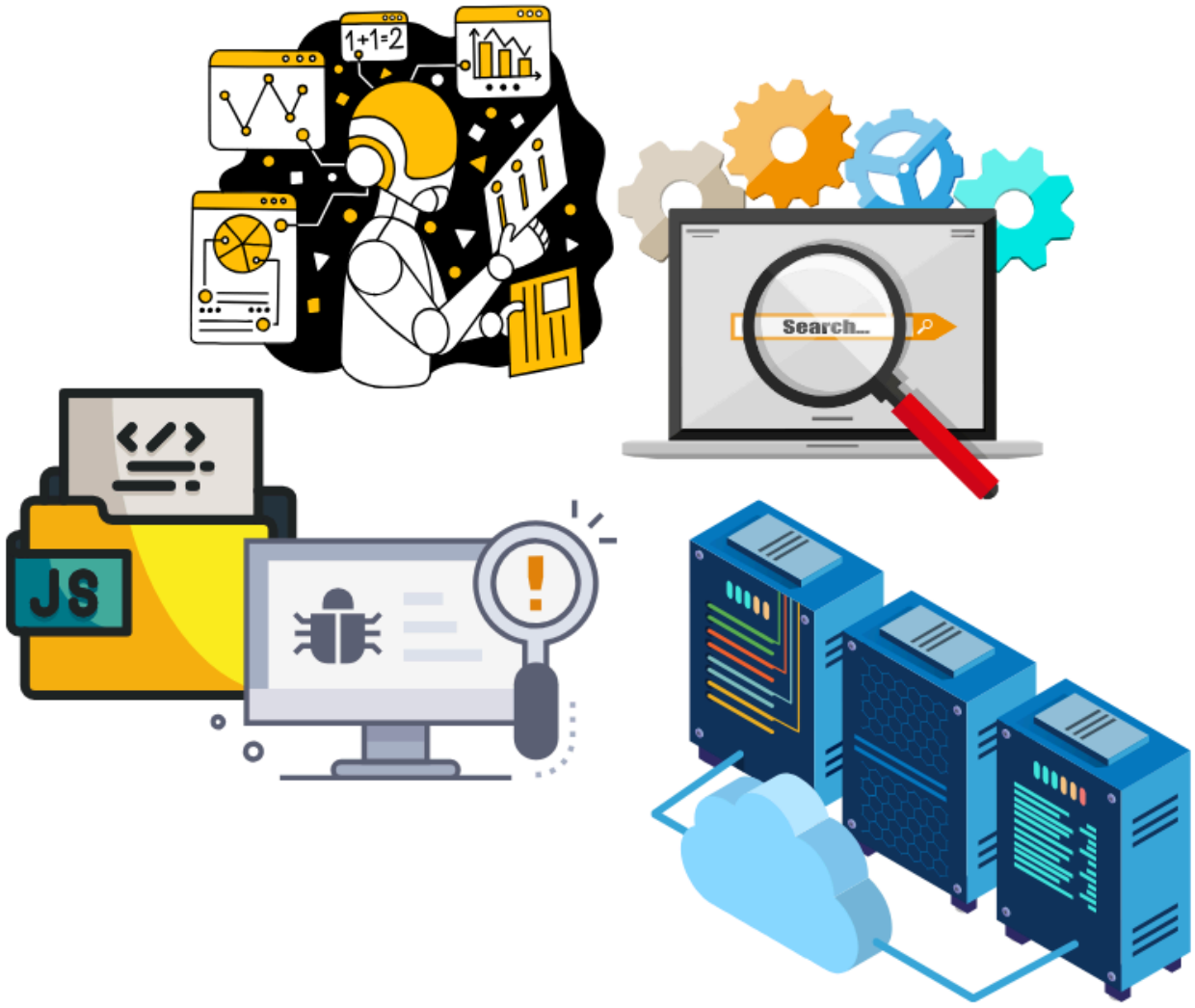
Coventry University

Softwarica College of IT & E-commerce

Design and Development of a Web Application to Detect and Analyse JavaScript Malware in an Enterprise Web Infrastructure Using Detection Engine and Machine Learning Techniques. (JS-MDA)

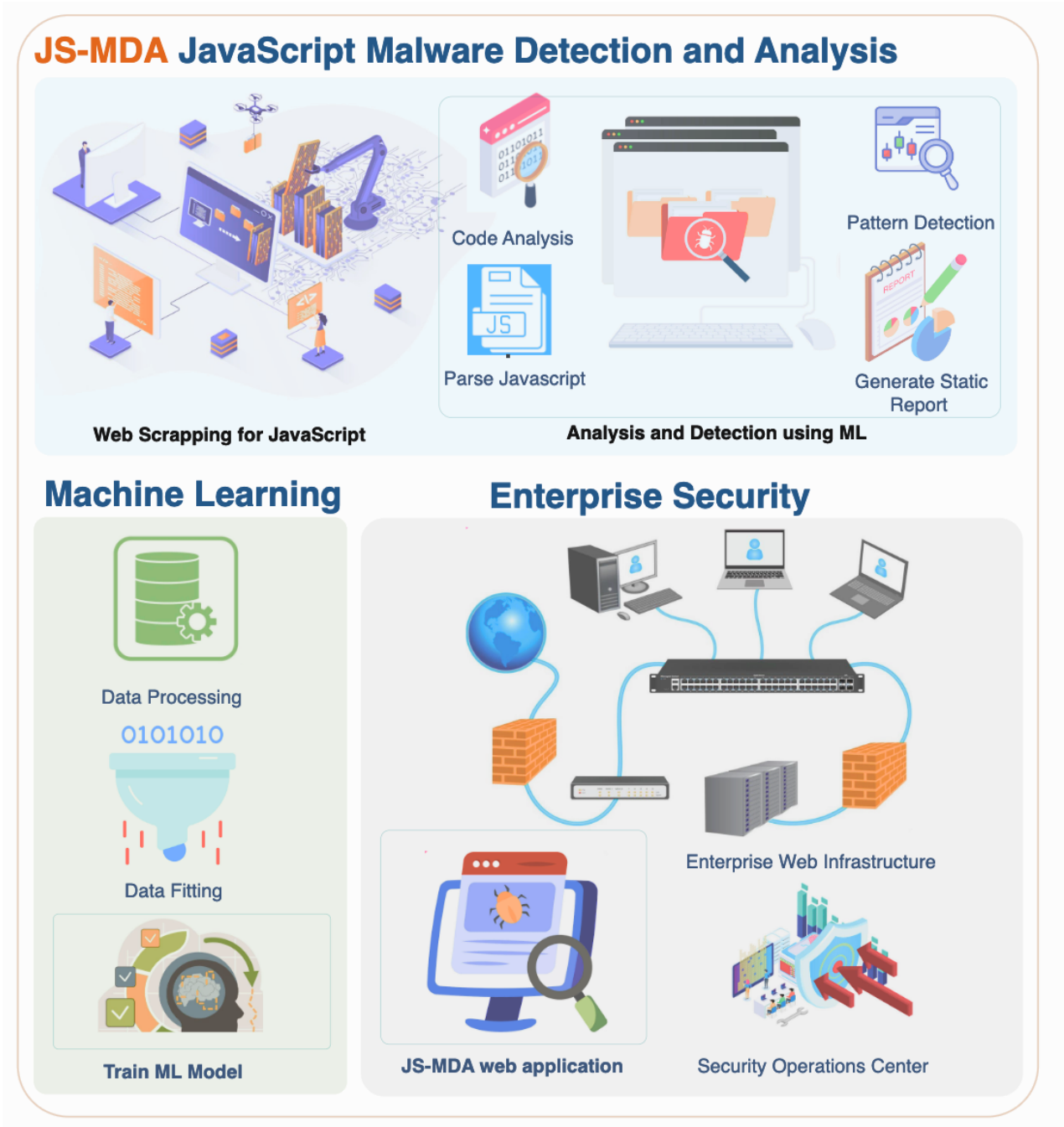
ST6047CEM CyberSecurity Project



Submitted by: **Rupak Rajbanshi**
Student ID: **210333**

Submitted To: **Manoj Shrestha**

CONCEPTUAL DIAGRAM



ACKNOWLEDGEMENT

My sincere gratitude is expressed to Mr. Manoj Shrestha, Head of Softwarica College of IT and E-Commerce, for providing me with an opportunity to learn structured thesis writing as well as guiding and organizing all the technical and research aspects of this thesis paper. I would also like to appreciate the efforts of my family, friends, and coworkers for their invaluable encouragement and psychological support during this endeavor, which contributed to the successful completion of the project. Moreover, I am grateful to the college and university for giving me such a valuable and worthy platform to commence my research paper, wherein I could discover knowledge and engage in endless learning.

ABSTRACT

Javascript is the most used programming language for client-side programming, and with its use come security issues. The most prominent one is the detection of javascripts on the websites. The antivirus application and the detection technique lack the ability to detect only JavaScript malware, as they are designed to perform generally on the applications, not the program that runs on the browser. Moreover, the browser-specific identification mechanism for detecting JavaScript malware is complicated because the web infrastructure has client and server sides. However, the requests and responses from the frontend and backend can be intercepted using applications like Burpsuite to extract the javascripts and do further investigation on them. The primary goal of the study is to investigate the detection of JavaScript malware using JS-MDA (JavaScript Malware Detection Application). JS-MDA is integrated with machine learning models and examined to distinguish between benign and malicious javascripts. Different ML models, such as Support Vector Machine, Naive Bayes, Singular Value Decomposition, Random Forest, K Nearest Neighbor, and Logistic Regression, are tested for their accuracy to detect javascripts based on the features. The features, lexical and syntactic, contain different levels of context and semantic information. In accordance with these patterns, each model is trained with malicious and benign javascript samples to improve the detection combined with the predictions. The pre-compiled, trained model is integrated into the backend of JS-MDA as a detection engine. Whereas the front end extracts javascript from the given url, then uses a detection engine to classify the javascript code as benign or malicious. Nevertheless, the identification of obfuscation and encryption can also be achieved prior to classification. The proposed solution depends on the MERN stack, Python Flask, and ML algorithms. Using these tools, technology, and infrastructure, the JS-MDA demonstrates effective detection and analysis of Javascript code.

KEYWORDS

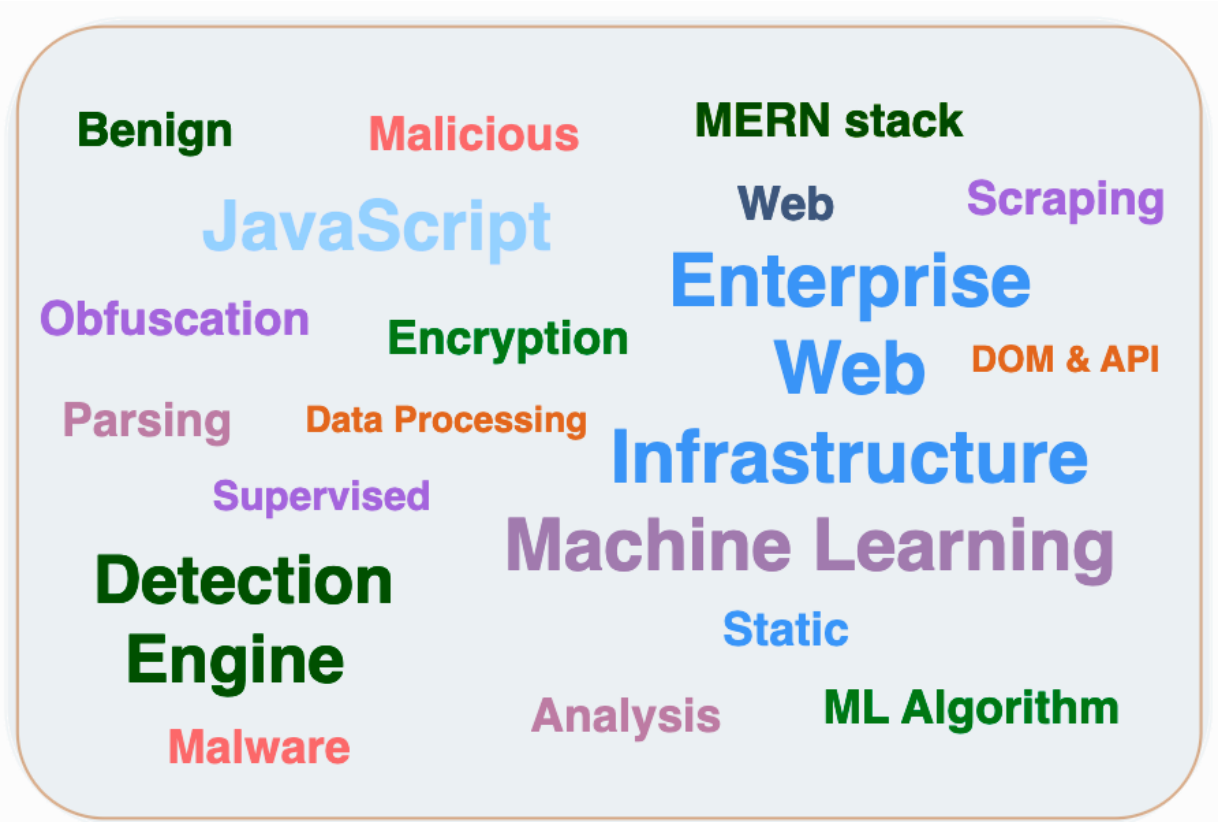

Benign
Malicious
MERN stack
Web
Scraping
JavaScript
Enterprise
Obfuscation
Encryption
Web
DOM & API
Parsing
Data Processing
Infrastructure
Supervised
Machine Learning
Detection
Engine
Static
Analysis
ML Algorithm
Malware

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

There are many enterprises, small or large, that use web platforms to provide services online. The services can range from entertainment, social networking, academics, news broadcasting, etc. With the adequate popularity and use of the platform, the web is also vulnerable to malicious actors, who may possibly compromise a client or the company. The actors can populate complex attack vectors to infiltrate web infrastructure and gain access to personal or business computers. Mostly, attackers try to manipulate javascript to export malicious code to the victim. The detection of these malicious scripts is technically not possible with antivirus, and the changing nature of the javascript has made its identification even slower. As a result, end users are prone to malicious attacks at any time of the day.(Muraya, *Web architecture and Infrastructure Design* 2023)

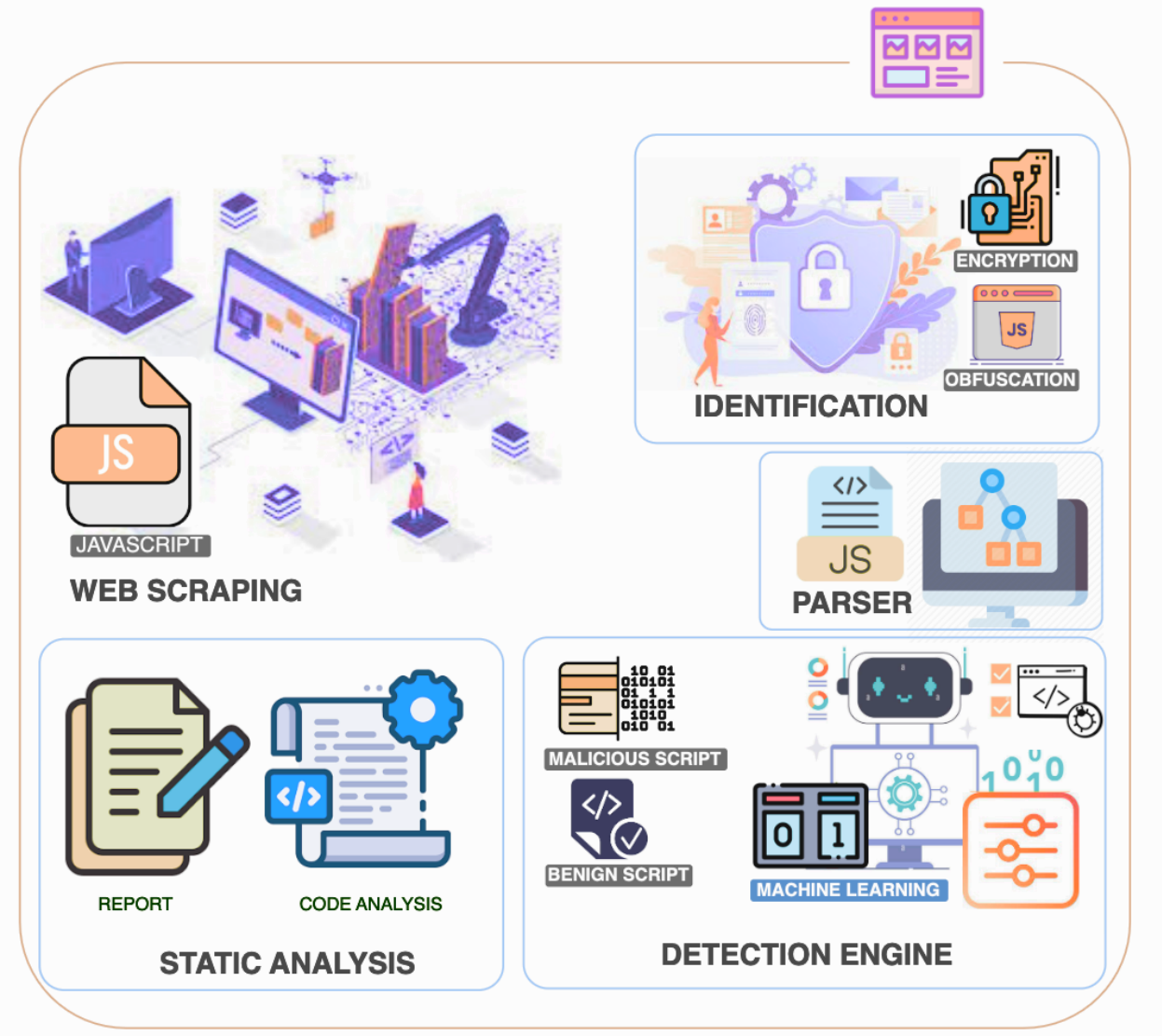


Figure 1: JS-MDA Overview

More than 95% of global websites utilize JavaScript and the framework. Javascript's use and structure across the web infrastructure provide extensive functionalities while also exposing its vulnerability (OWASP 10). One of them is DOM (Document Object Model).

(Academy, *Dom-based vulnerabilities* 2024) DOM manipulation with javascript is a mostly used feature, with potentially dangerous javascript handling methods, source, and sink. The javascript property source can be controlled to execute a sink function arbitrarily; for example, redirect the victim to a malicious site. Despite the evolution of various techniques over the years, it is still difficult to detect malicious javascripts, and perhaps we are uncertain about completely eliminating DOM-based attacks.

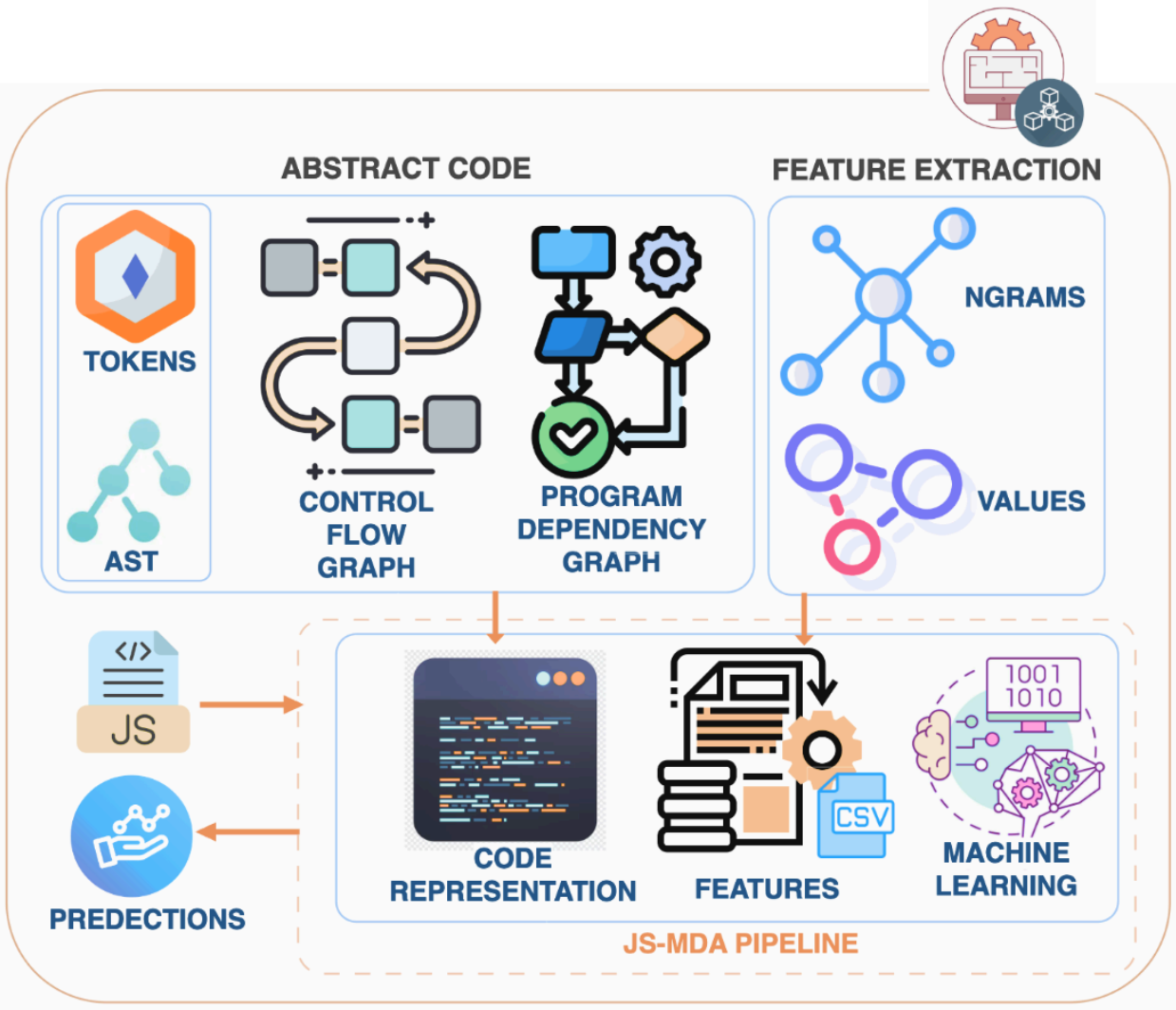


Figure 2: JS-MDA Module Architecture Overview

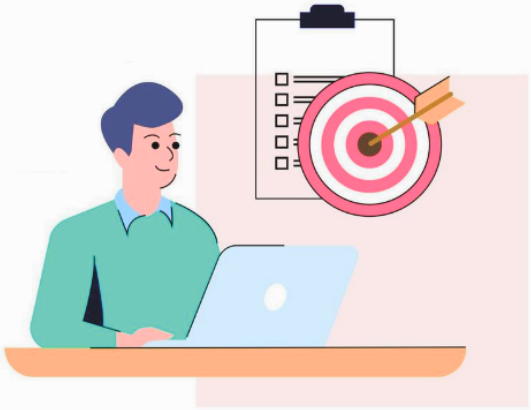
However, it may be feasible to identify web sites containing malicious javascripts, but the key challenge comes in distinguishing scripts as malicious or benign since a lot of the time code may be discovered obfuscated, which restricts the detection scope. Therefore, this study effort focuses on the limits that give insight into combining the detection engine with machine learning approaches. JS-MDA is a method that enhances detection capability by breaking down the code into its most basic and conceptual parts (syntactic and lexical levels) in search of recurring and specific traits that may be benign or malicious. It is a machine-learning-integrated detector designed to identify malicious JavaScript scripts.

In this work, the JS-MDA technique is expanded further to undertake research on the AST (control flow information) pipeline to process javascript based on the syntactic and semantic order of the code. The syntactic order specifies the statement blocks, expressions, control structures, function declarations, variable declarations, objects, and arrays. Whereas semantic order pertains to hoisting, scope and closures, binding, prototype chains, event loops, and asynchronous programming, The syntactic procedure is to produce an AST (abstract syntax tree) by analyzing the token sequence from lexers to identify the grammatical structure of the code, and the semantics help comprehend the meaning and behavior of the code. The core features of JS-MDA are feature extraction, a Javascript classification model, and a static analysis report.

The classification model is assessed on datasets including 3,000 benign scripts and 5,000 malicious scripts. The major emphasis is on each model's capacity to detect benign and malicious javascript code. To increase the accuracy of the classifier, SMOTE and PCA tuning is paired with model training. The model is assessed on precision, recall, F1-score, confusion matrix and the AUC score.

AIM

Aim of JS-MDA



" Design and develop an application JS-MDA (JavaScript Malware Detection and Analysis tool) to understand the nature of javascript malware by analysing and detecting it, using detection engine and machine learning techniques, to leverage the defensive security of Enterprise Web Infrastructure

Figure 3: AIM

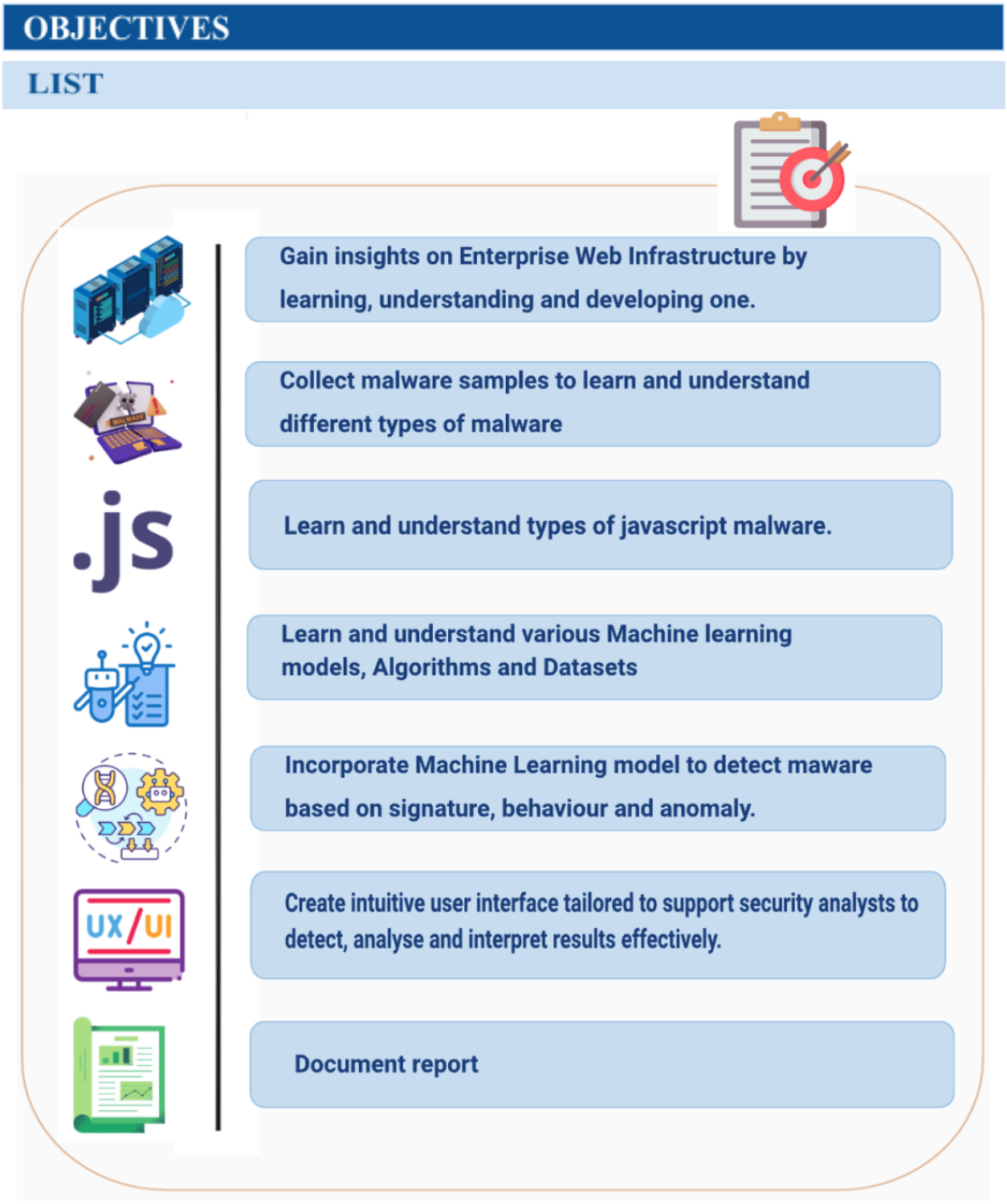


Figure 4: Objectives List

JUSTIFICATION

The digital landscape is ascending everyday with the increase of new Business or Enterprises. With expanding use of the web technology, protecting essential systems and digital assets has become more difficult than ever before. This typically includes hardware and software assets such as data center resources, end devices etc. As more business activities are done digitally, the data transactions done using public unsercure networks to the corporate networks creates many vulnerable opportunities for the threat actors. However, there are enterprise IT infrastructure security frameworks that most of the organizations address. The framework defines the level of security for Data, Application, Network and Hardware. All the four areas of security further demand vast understaning and expertise. So the scope of this paper is more focused towards Application security. (Infrastructure Security, 2024)The web application security is the most crital part to acknowledge, as most of the web attacks are implemented through the applicaiton layer. Many web applications come with a lot of security vulnerabilities. These security vulnerabilities are further described in OWASP(Open Web Application Security Project) top 10. The web applications are teseted on these listed security vulnerabilities. Nevertheless, the vulnerabilites can be identified but not all can be mitigated completely, as almost all the data traffic is generated from the client side, for big corporate web applicaitons, there maye be some weakness inthe application exposed to the world. These vulnerabilities can contribute towards exploitation, to leak sensitive information and compromise the application systems. A primitive and common approach to detect these types of web attacks are known as IDS (Intrusion Detection System). On the contrary, the evolving nature of the technology and the development of sophisticated attacks can easily bypass IDS systems. For example, a malicious javascript attached to htlm request can be identified by the IDS but, the same script morphed or obfuscated makes detection complicated.

TRADITIONAL MALWARE DETECTION PROBLEM

The majority of malware detectors can be seen signature-based following the present day. A web is a large-scale environment where old methods of malware detection have less potential. A basic malware detector uses a large set of databases with an extensive list of known signatures and an algorithm to check a file or a code to check against those verified patterns. The process is resource intensive and time consuming, as a result the detection of any file would take much more time than required. The limitation is only signature based, a new unique pattern is bound to be updated in the database, which can create significant delays for updates in the larger organizations. The heuristic analysis approach provides only basic information and lacks deep behavior analysis of any malware detected. Moreover, the running environment for old malware detectors and malicious javascript is completely different, as javascript completely runs on the browser. The browser provides a runtime environment for javascripts, to take place interactions with web APIs and DOM. The working environment of javascript is flexible and equipped with many features. Due to the understanding restrictions of the environment, the traditional malware

detectors cannot identify changing behavior of the javascript. To this present day, the techniques to manipulate and hide javascript have evolved from many generations, the modern methods include, obfuscation, encryption, polymorphism, metamorphism. These methods are very prominent and can evade detection systems easily. The tactic is each time the code executes, the appearance of the code changes which hides the intent of the code. Not only this, the dynamic code execution is a feature that executes code during runtime to evade signature based detection. Therefore, to this point it is clear that the javascript itself is very advanced and progressive with time and not every malware detectors can identify malicious javascript without any struggle. With time new variants of javascript will be even more difficult and require complex methods of detection and analysis. (D.Gantz & R.Philpott, 2013)

Figure 5: Traditional Malware Detection Problem List

Figure 6: List of Solutions

In contrast, javascript malware detection is a matter of significant concern, the existing technologies and tools can comprehend the mitigation strategy to some extent but they are unable to provide complete security to the web framework. The proposed method involves the integration of Machine learning to effective detection and analysis of benign or malicious javascript in the web browser environment. Therefore, the web scraping module for the given urls is tuned to scrape for javascripts. The javascripts undergo additional parsing to extract features and tokens. Based on the raw data feature engineering is carried out to create new features that are more informative and well suited for machine learning. The ML model is optimized using hyperparameter tuning to find the most optimal configuration to maximize overall performance of the model. With the implementation of these trained models, enables faster identification of malicious javascript. Moreover, the pre-trainied models are straightforward to customize with minor and periodic updates to further elevate the understanding of unfamiliar patterns or pattern behaviors. The classifiers recognize the scripts through the examination of both lexical content and syntatic structure. Furthermore, following identification of the scripts as benign or malicious, the static analysis feature shows the key facts in a more meaningfull format. The static analysis result presents clear insight into the occurrence of the lexical and syntactic characteristics, while the additional information about the suspicious patterns suggesting obfuscation , dynamic code execution and network requests enhance the quality of the report.

By implementing the proposed JS-MDA (JavaScript Malware Detection and Analysis tool) to the enterprise, the security analyst can further investigate the malicious intent of javascript. The IDS and JS-MDA module combined can leverage the detection and static analysis capability. This research study supports the integration of the JS-MDA to the relevant web architecture. JS-MDA is a web application that is designed to allow small scale businesses to move one step further towards web security following other developments in the cybersecurity domain.

Figure 7: Research Questions

Figure 8: Scope - Problems, Solutions, Functions and Tools

ETHICAL CONSIDERATIONS

The learnings and discoveries illustrated in this paper acknowledge the effort of the author of the research materials through proper citation and reference preserving academic norms.. The contents include conference papers, research papers, journals, books and trusted sources. In the context of presenting this research, ethical concerns were maintained ensuring no harm to any organization or at personal level. Understanding data protection and privacy issues, sufficient authorization was taken for data handling obliged to adhere within legal and regulatory boundaries. Similarly, the secondary sources of softwares were studied only for learning purposes, no purposefull fabrication of any comparable product, promoting financial and personal dispute, the project fostered to protect copyright regulations. Legitimate research was undertaken to design and develop the project JS-MDA, utilizing open source technology and tools from trusted sources. The CIA triad principles were the standard and obliged to be in conformity with the Data Protection Act and Electronic Transaction Act of Nepal.

Figure 9: Ethical Considerations

Figure 10: Research Methodology

Based on the desk based research method, the complete concept to design JS-MDA formulated from analyzing working mechanisms of antivirus software, IDS, malware analysis tools etc. These tools and technology span the entire spectrum of Information Technology but the major purpose was to deep dive into one trending technology and framework, and provide a reliable solution dependent on that. Choosing Www (world wide web) the most prominent technology

and an idea to provide web security in one specific way came foremost. The web technology employs javascript and malicious javascript was in existence from the beginning when javascript was launched *(Maricheva, 2023)*. And JS-MDA is a system to identify and analyse malicious javascript.

The desk based study was divided into looking for diverse publications and obtaining skill based on understanding. The relevant subject topic was enough to discover broder range of necessary resources. The book based materials were structured for learning, to gain actual practical abilities to grasp web architecture, technologies and programming languages. Whereas, the academic journals, research papers, conference documents delivered an insight on how different technologies have been implemented to develop applications to complement existing framework operating on the web environment. Moreover, the literature research also provided information on the challenges and requirements specific to the development of JS-MDA, through combined assessment of real-life security breaches and relevant tools required to provide mitigation strategy.

Furthermore, despite the fact that this research approach is time demanding, the matter of overall experience on research study is enough to generate clear concepts on the subject and derive proper plan towards designing and development of the application. The way of conducting research turned out to be quite efficient, productive and the timeframe covers all aspects gracefully with great findings.

CASE STUDIES

The case studies on the relevant subject area are depicted into three categories. The primary purpose to split the study scope is to see the problems through different perspectives and define the strategy to complement the design and development of the web application. The first category is to findout browser vulnerabilities and extend research only specific to javascript. The second is to connect those vulnerabilities to resulting real life security breaches. And third, learning from the pattern of the breach, do further research on the detection mechanisms and tools that are capable of mitigating specific types of vulnerabilities through prior detection to breach and provide useful insight on the exploits using machine learning techniques.

The factors that contribute javascript application to vulnerabilities is the client-side execution and Node.js for server-side development. The threat actors can manipulate the browser activities to exploit these vulnearbities. The complex javascript ecosystem, insecure coding practices, dynamic nature of javascript and lack of built-in security features leave javascript vulnerable. The client-side vulnerabilities can range from Cross-Site Scripting(XSS), Cross-Site Request Forgery (CSFR), Javascript Hijacking etc and server-side vulnerabilities can be arbitrary code injection, command injection, improper error handling, iserucre deserialization, unvalidated redirects and forwards, insecure use of third party libraries and components. (OWASP Foundation, 2022)

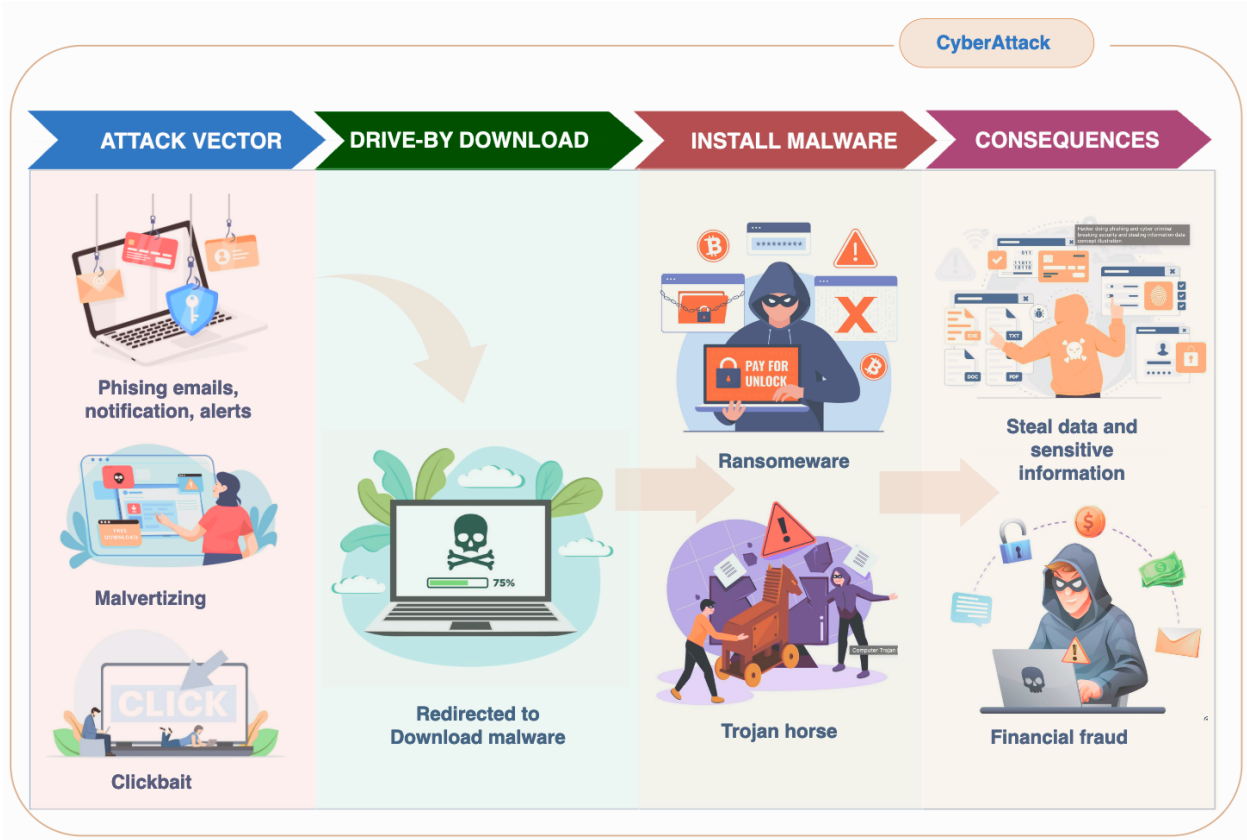


Figure 11: Cyber Attack Process

These vulnerabilities are exploited using different attack vectors that lists Phishing, Malvertising, XSS, SQL injection, Ransomware, trojan horse, MITM, drive-by-downloads etc (*Server-side vulnerabilities* 2024).Most of the time the threat actors use hybrid methods example: use of phishing to initiate drive-by downloads. The most prominent vulnerabilities exploited through web browser, plugins and scripts is to redirect download and install malware silently on the target is called drive-by downloads. This attack vector combined with others can be the most sophisticated form of attack and have compromised many big companies in the past.

Moreover, there are many cases that have involved javascript and drive-by downloads to compromise the security posture of the company. On November 4th 2012, the U.S broadcaster NBC website was hacked delivering malicious software intended to steal user banking information.(Brewster, 2013) The exploit citadel banker trojan vulnerability was dropped that is used for banking fraud and cyber-espionage (*CVE-2013-0422* 2024). The trojan was delivered via exploiting iframe to redirect links to download malware using drive-by downloads. (Perez, 2013)

In addition to this, Flashfake, a Mac OS X trojan, infected 700000 computers worldwide(siliconrepublic, 2012). Google's ad revenue was used, trojan's ad-clicking component was used to intercept browser requests and redirect to bogus sites, using javascript code to load the exploit, download and run malicious code remotely. (Reed, 2024)Another advancement to this was trojan botnets.

Not only this, November 2014, the malvertising campaign targeted yahoo and affected 2 million yahoo customer pc with malware. This time the adobe's flash vulnerability was used through a fraud ad campaign to install malware for ransomware programs. (Hern, *'Malvertising' campaign* 2015). The most used pattern observed in breach action is to deliver and download malware to the victim.

Furthermore, the compromises were from the early days when javascript was new and its usage was growing day by day. In that time, due to the nature of the javascript there were many vulnerabilities with the javascript platform itself and not many detection mechanisms. However with time, different detection and analysis mechanisms have developed and among them some are discussed and included as a part of this research studies. CUJO is embedded in a web proxy that is specifically designed to detect drive-by download attacks. A JSTAP is a static malicious javascript detection system and a ZOZZLE is a fast and precise in browser javascript malware detection and prevention system. The working mechanism of these tools are studied and presented review based on through analysis. The analysis effort was focused to findout the efficiency of these modern tools to accurately identify vulnerabilities and provide analysis results. Even though the purpose of each tool is to work with javascript, no tool is designed the same way. The working mechanism is integrated with machine learning techniques but the scope of analysis is vast and different from one another.

Therefore, the research has provided enough opportunity to clear the concept behind each tool, vulnerabilities and understanding problems from the impact of real life breaches further paving the way to brainstorm supporting ideas to design and develop web applications designated to detect and analyse javascript malware which is the main objective of this research paper.

CUJO

Javascript is a core component of websites that offers several functionalities and limitations. Besides, it may be used to exploit vulnerabilities present in online systems. Drive-by-downloads are a very effective technique for exploiting web browser vulnerabilities and associated extensions. The extension's security vulnerability may be used by the attacker with evasive javascript to silently download and install malware without detection. The virus may take the shape of a keylogger, ransomware, spyware, or similar malicious software. Its primary purposes may include stealing financial information or transactions, encrypting data and demanding ransom, monitoring activities, and collecting sensitive information. (Rieck et al., 2010)

Nevertheless, there are detection methods such as CUJO (Classification of Unknown Javascript code) that may effectively counter these kinds of incidents. It is a web proxy plugin that includes an automated detection mechanism designed to avoid drive-by-download assaults.(Duncan, 2022)CUJO incorporates both static and dynamic analytic principles. In simple terms, it prevents the transmission of malicious javascript code from the client to the server. Additionally, it employs machine learning techniques to provide analytical capabilities for detection models. The feature extraction is based on the attack characteristics and analysis patterns.

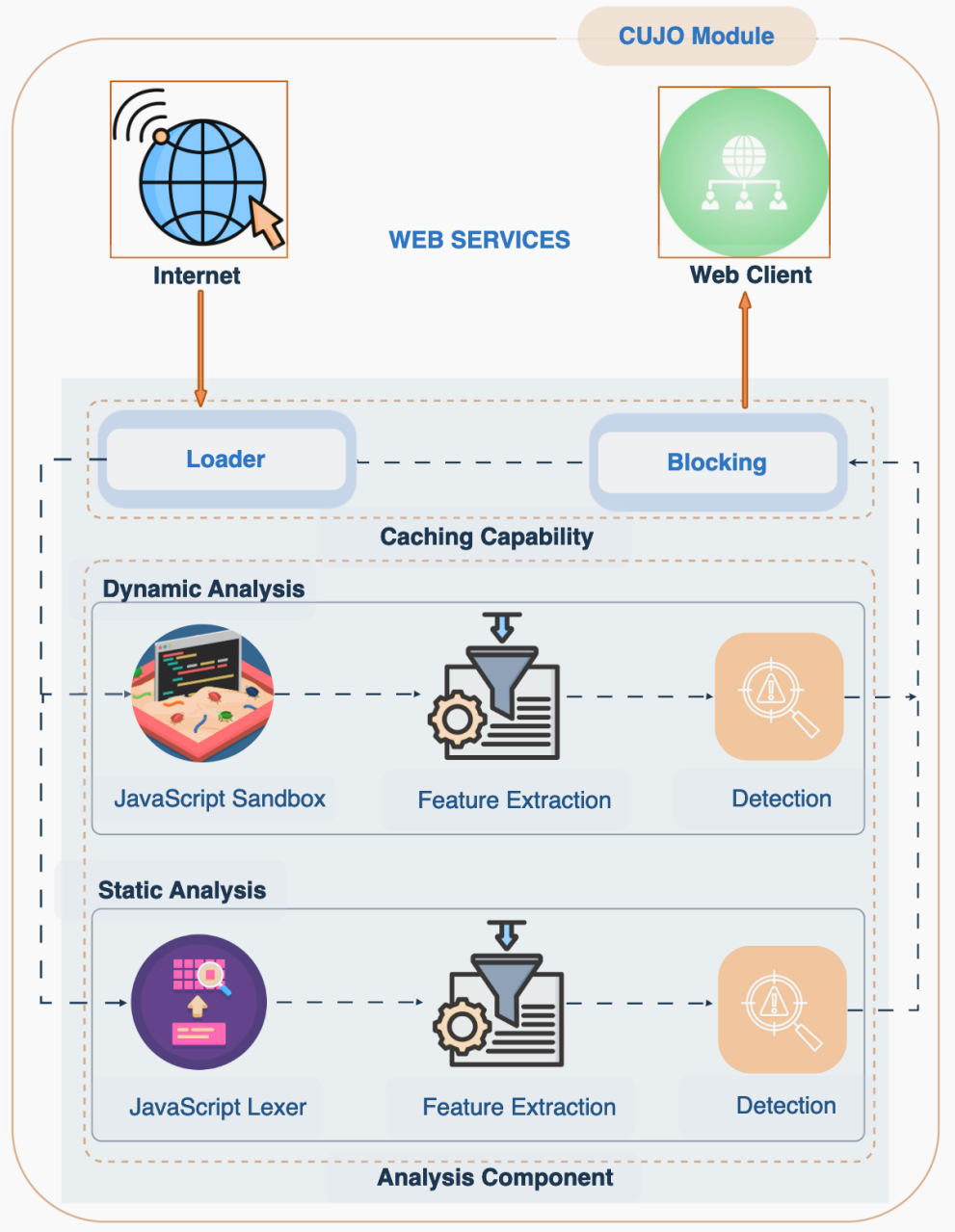


Figure 12: Schematic Depiction of CUJO

Furthermore, CUJO adopts a caching technique to store analyzed data i.e malicious javascript code from web sites and its results. For deep malware analysis, the code is analyzed to extract lexical tokens using YACC grammar. To facilitate static analysis the STR and NUM encoding is done to represent numericals and strings, and afterward length of STR or NUM and function analysis keyword is encoded. So, when string operation is done to call any function, the pattern related to the drive-by-download attack is recognized. In addition to this, dynamic analysis adds abstract operations for recognized patterns, such heap-spraying attacks and usage of browser functionalities.

Moreover, the static and dynamic set of findings are represented using the q-gram concept, *((N-grams* 2024) subsequently to map analytical reports into vector space and utilizing efficient data structures to sort arrays of reports to represent it into vector space. The features are retrieved as per the report produced from static and dynamic analysis. These features acquired from multiple attack patterns and reports from benign javascript code are then applied to train SVM (support vector Machine).

Attack data sets	CUJO		
	static	dynamic	combined
Spam Trap	96.9%	98.1%	99.4%
SQL Injection	93.8%	88.3%	98.3%
Malware Forum	78.7%	71.2%	85.5%
Wepawet-new	86.3%	84.1%	94.8%
Obfuscated	100.0%	87.3%	100.0%
Average	90.2%	86.0%	94.4%

True positive *(datasets)*

Benign data sets	CUJO		
	static	dynamic	combined
Alexa-200k	0.001%	0.001%	0.002%
Surfing	0.000%	0.000%	0.000%

False positive

The outcome of detection based on true positive and false positive rates of CUJO on the benign datasets and attack datasets demonstrates that the identificaiton rates are better with combined analysis. The identification on attack datasets provides the analysis on true positive rate, with an average of 94.4% combined, confirming successfull detection of most of the drive-by-download attacks. The 0.001% false positive for both static analysis and dynamic analysis occurred partly due to the encryption of the javascript, suggesting existence of undetected vulnerabilities in the benign dataset.

Contribution $\phi_s(x) \cdot w_s$	Features $s \in S$ (4-grams)
0.044	`+ STR.01 , STR.01`
0.043	`WHILE ( ID .`
0.042	`= ID + ID`
0.039	`{ TRY { VAR`
0.039	`) { } }`

(a) Top q-grams of a heap-spraying attack

Contribution $\phi_s(x) \cdot w_s$	Features $s \in S$ (3-grams)
0.190	`HEAP SPRAYING DETECTED`
0.121	`CALL unescape SET`
0.053	`SET global.shellcode TO`
0.053	`unescape SET global.shellcode`
0.036	`TO "%90%90%90%90%90%90...`

(a) Top q-grams of a heap-spraying attack

Static and Dynamic Detection analysis using q-grams

Considering the following table as an example, the report is a static and dynamic analysis of two drive-by-download attacks, proving the patterns and q-grams characteristics contributing to the categorization of the attack type. Tokens extracted are utilized to figure out the nature of the attack and the numerical value of q-grams represnts the association with this specific form of attack. The accuracy to detect malicious patterns heavily depends on the identification of q-grams which is primarily related to the vulnerability in the browser extension. Despite the fact that, the implementation of CUJO does not completely eliminate the drive-by-download threat, but it offers a comprehensive technique of static and dynamic detection model that detects the malicious code and prevents it prior to delivery.

ZOZZLE

ZOZZLE is a static javascript malware detector that is integrated within the browser's javascript engine. It collects and processes runtime javascripts i.e before javascript gets executed, ZOZZLE hooks the runtime and exposes the malicious content by analyzing javascript *(Richards et al., 2011)*. It utilizes a NOZZLE runtime detector to collect an extensive list of javascripts classified as malware and benign javascript samples from the wild scanning URLs. AST-based detection method is applied, using a context-sensitive feature approach for feature extraction. These attributes are utilized to train the Baysian classifiers and use the learned classifier to distinguish new javascript contexts as malicious or benign.

Moreover, In the internet explorer browser the javascript engine employs binary instrumentation to intercept calls to the compiler function found in the jscript.dll library. This function is referred to as code context and is invoked when 'eval' is called. In this manner javascript is assessed before run by the engine. For the code context, features are extracted based on the hierarchical structure of javascript. The quantity of occurrences are not recorded, but the appearance of the features that matters.

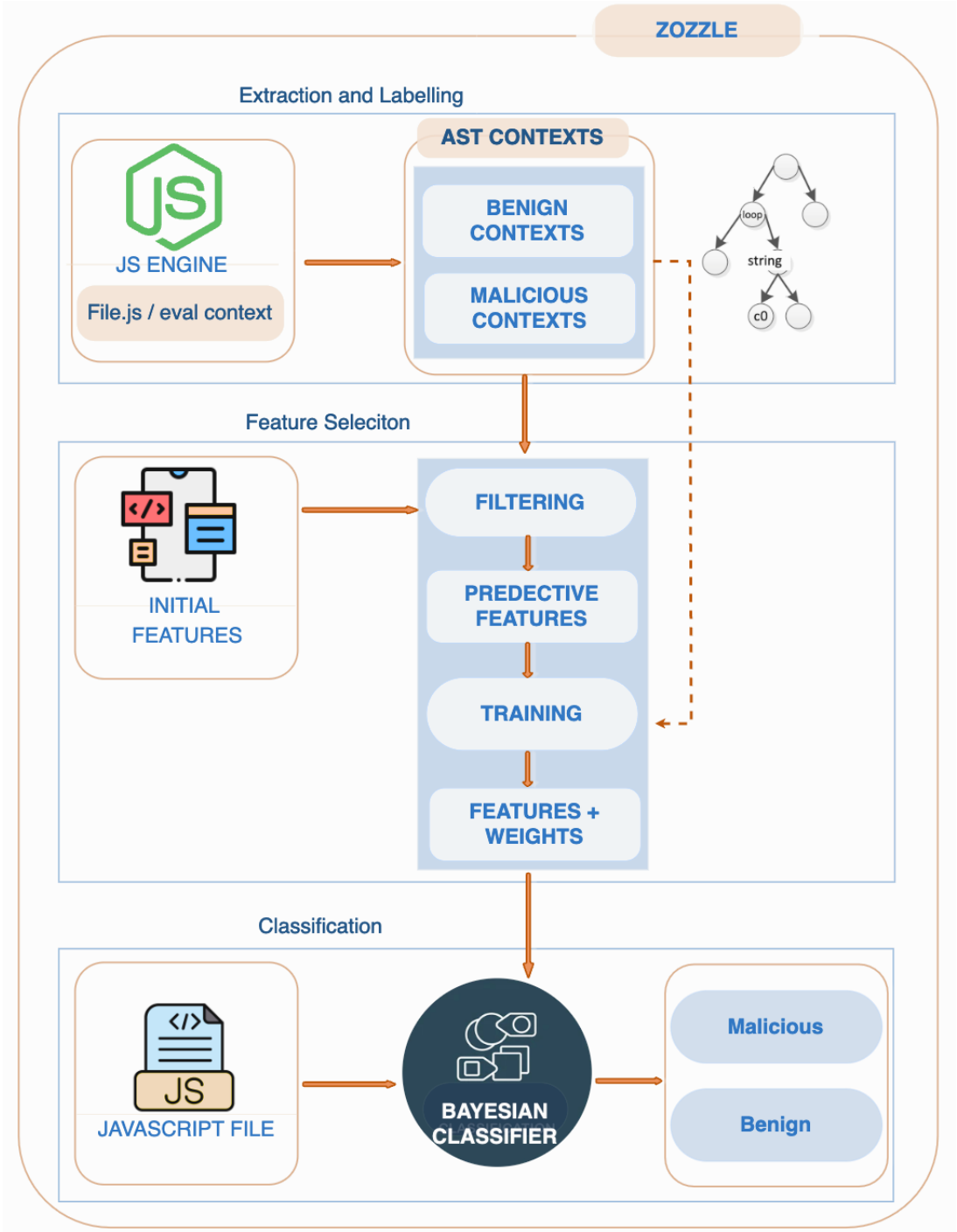
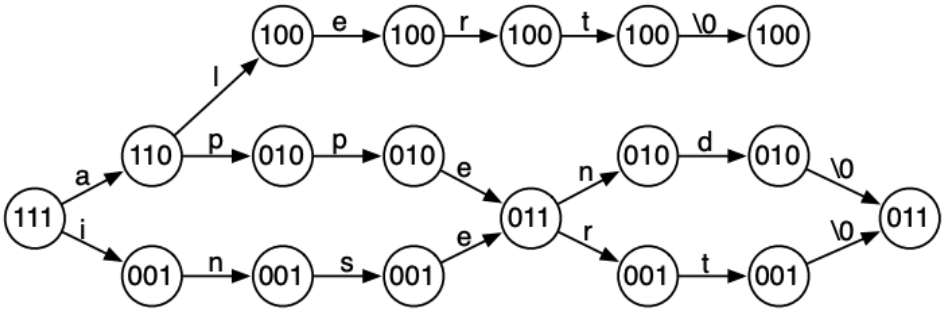


Figure 13: ZOZZLE Training Process

The distinctive features are obtained from the expression and variable declaration nodes. Pre-selected features are utilized to train classification which are informative and correlate with grouping the script as malicious or bening. Fast pattern matching is used to select similar traits based on a bitmap set to the occurrence of the character present in the node.



Fast Feature Matching

(Curtsinger et al., 2011)

Furthermore, the performance of the classifier is evaluated depending on the specific type of dataset chosen during feature selection. The features comprise two sets of training datasets that are manually recognized sets of javascript code and automatically detected by NOZZLE. The manual method was to hand-pick the malicious javascript and the automatic way was to choose both sorts of features. However, benign javascript yield was more significant incompared to malicious. The section of AST context contains flat features that refer to the text of javascript code and hierarchical features (1-level and n-level) carrying contextual information.

Features	Hand-Picked	Automatic	Features
flat	95.45%	99.48%	948
1-level	98.51%	99.20%	1,589
n-level	96.65%	99.01%	2,187

Features	Hand-Picked		Automatic	
	False Pos.	False Neg.	False Pos.	False Neg.
flat	4.56%	4.51%	0.01%	5.84%
1-level	1.52%	1.26%	0.00%	9.20%
n-level	3.18%	5.14%	0.02%	11.08%

Feature Extraction

The accuracy of the classifier is evaluated depending on the false positive and false negative outcome. The table shows the false positive rates of handpicked features are considerable as compared to the automated feature selection. Similarly, the false negative rates are highere for

automated feature selection than handpicked features. Among all 1-level classifiers it is more effective with modest false negativee rate and extremely low false positive rate.

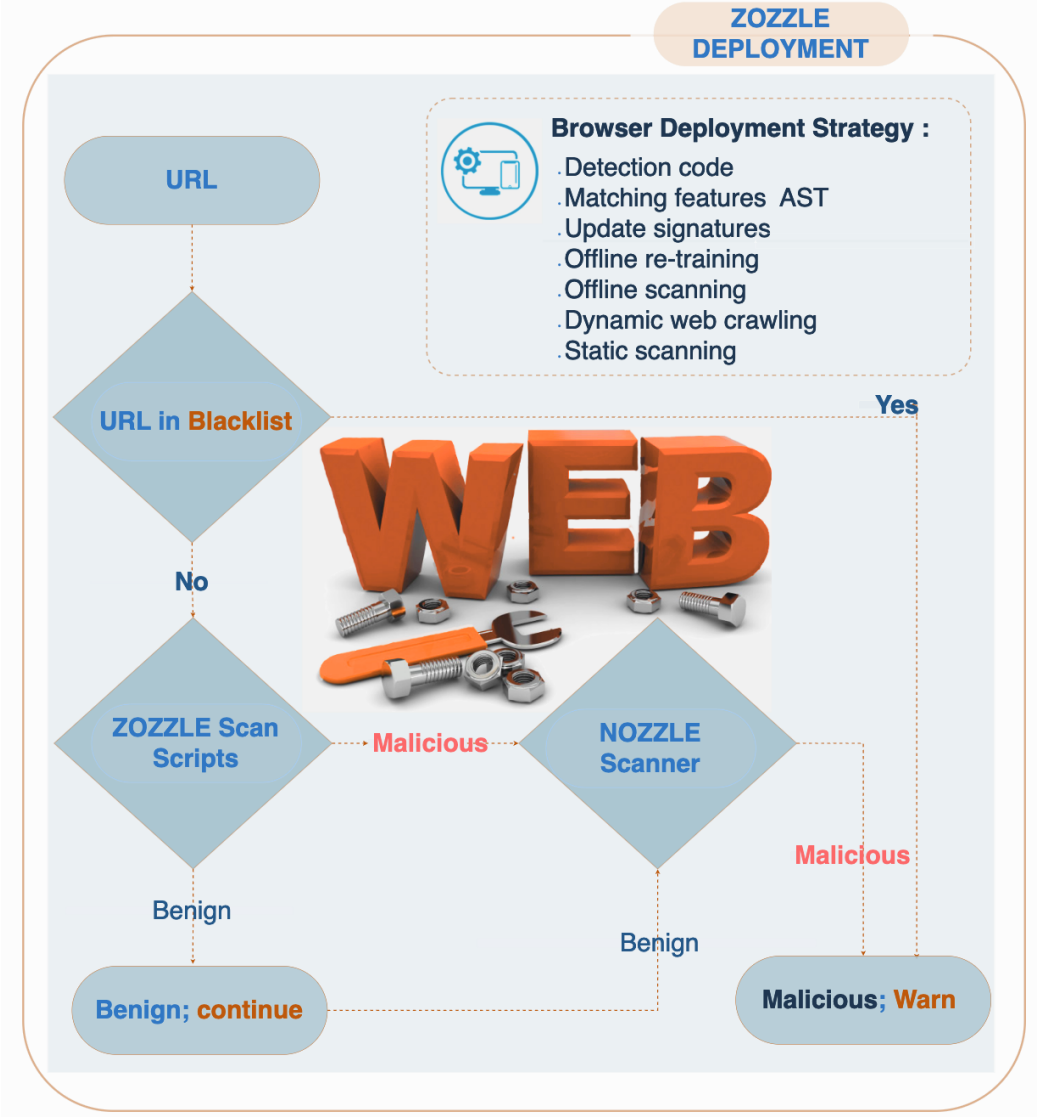


Figure 14: ZOZZLE In-browser Deployment

Therefore, the ZOZZLE is effective to statically detect any attempted exploits as it incorporates semantic information from the javascript during analysis. The adaptation to complementary detector NOZZLE for continuous update and training on collected malware samples, the efficiency of ZOZZLE to identify javascript malware improves with less human effort. The classifier re-training may be done offline, merely to update the new signatures of malicious javascripts. The in-browser deployment of ZOZZLE streamlines the sophisticated detection approach into more practicable, for in context or any static javascript code analysis. Therefore, ZOZZLE is a fast and accurate malware detecting tool that is intended to be effective in the browser environment.

SYNTACTIC DETECTION OF MALICIOUS JAVASCRIPT JSTAP

It is a modular system designed to identify JavaScript. It consists of 10 modules, which provide five distinct methods of code abstraction and two methods of feature extraction. An analysis tool supplied by the user that includes lexical analysis, abstract syntax tree (AST), control flow graph (CFG), program dependence graph (PDG), and data flow and control information. The analysis functionality may be customized to do analysis using either n-gram or by including variable information. The practical implementation yields good accuracy for each classification module, whether operating in combination or alone. The JSTAP module's architecture comprises code representation, feature extraction, and learning components. The frequencies derived from each javascript sample are used as learning components to differentiate between malicious and benign.

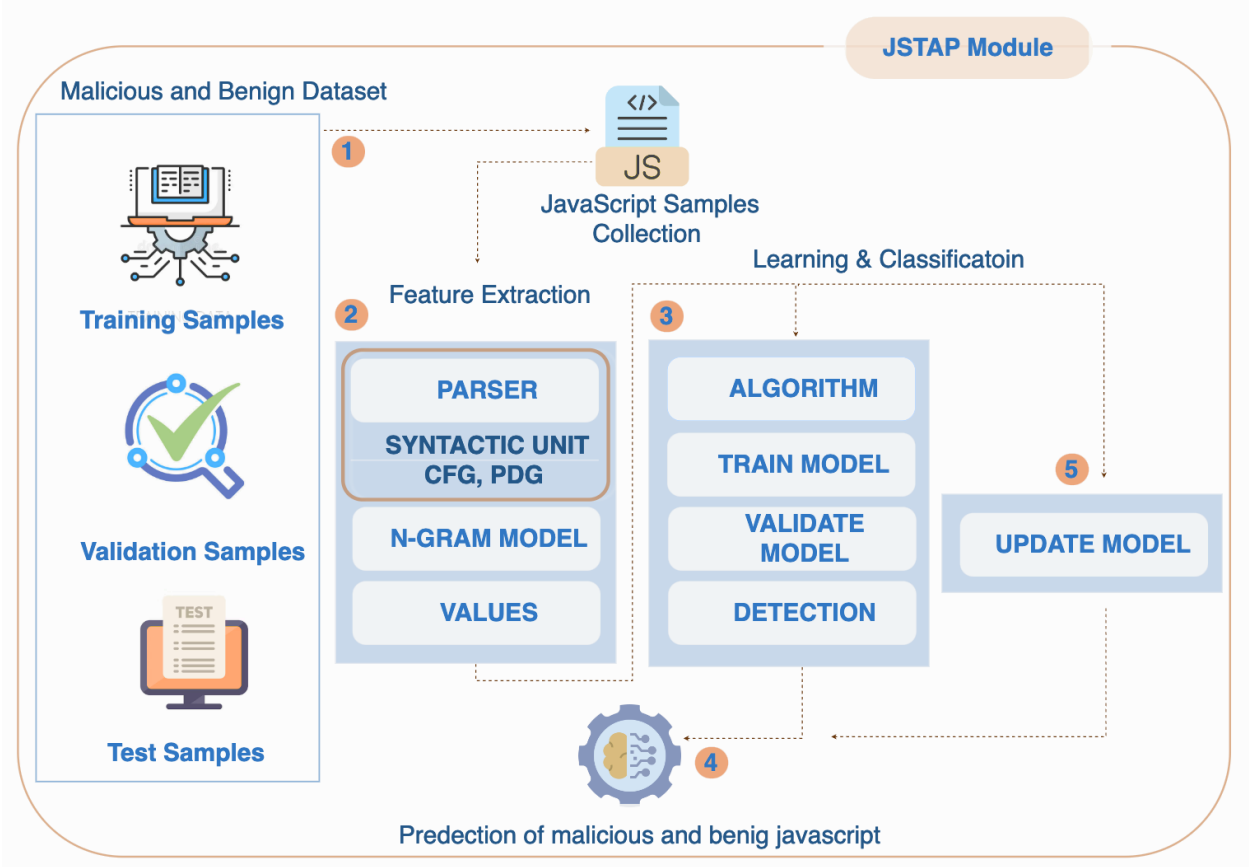


Figure 15: JSTAP Module Architecture

In addition, feature extraction involves various levels of abstraction in JavaScript code, including lexical analysis (tokenization) and the Abstract Syntax Tree (AST) that represents the program's grammatical structure. The Control Flow Graph (CFG) captures semantic information related to conditions, while the Program Dependence Graph (PDG) represents the program's execution order. When combined, these elements represent the properties of the code.(Fass et al., 2019)

Code example :

```
x.if = 1;
var y = 1;
if (x.if == 1) {d = y;}
```

Token	Value	Token	Value	Token	Value
Identifier	x	Numeric	1	Punctuator	)
Punctuator	.	Punctuator	;	Punctuator	{
Keyword	if	Keyword	if	Identifier	d
Punctuator	=	Punctuator	(	Punctuator	=
Numeric	1	Identifier	x	Identifier	y
Punctuator	;	Punctuator	.	Punctuator	;
Keyword	var	Keyword	if	Punctuator	}
Identifier	y	Punctuator	==		
Punctuator	=	Numeric	1		

Lexical Units extracted from code example

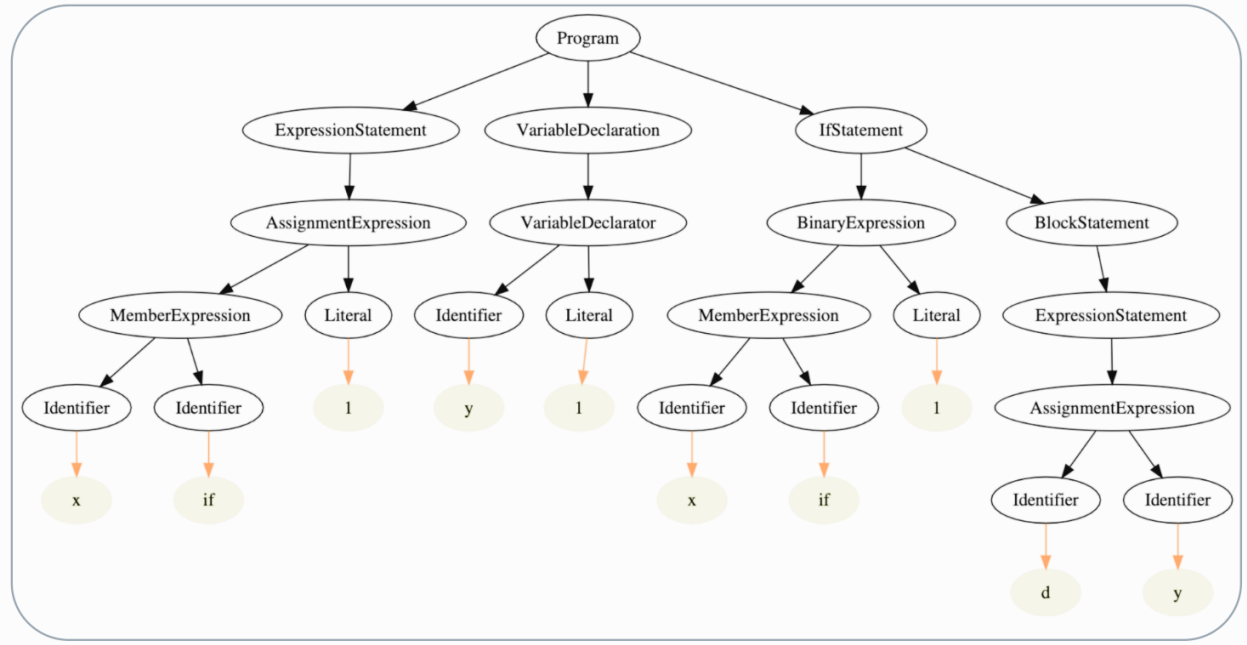


Figure 16: Control Flow Edges CFG with AST

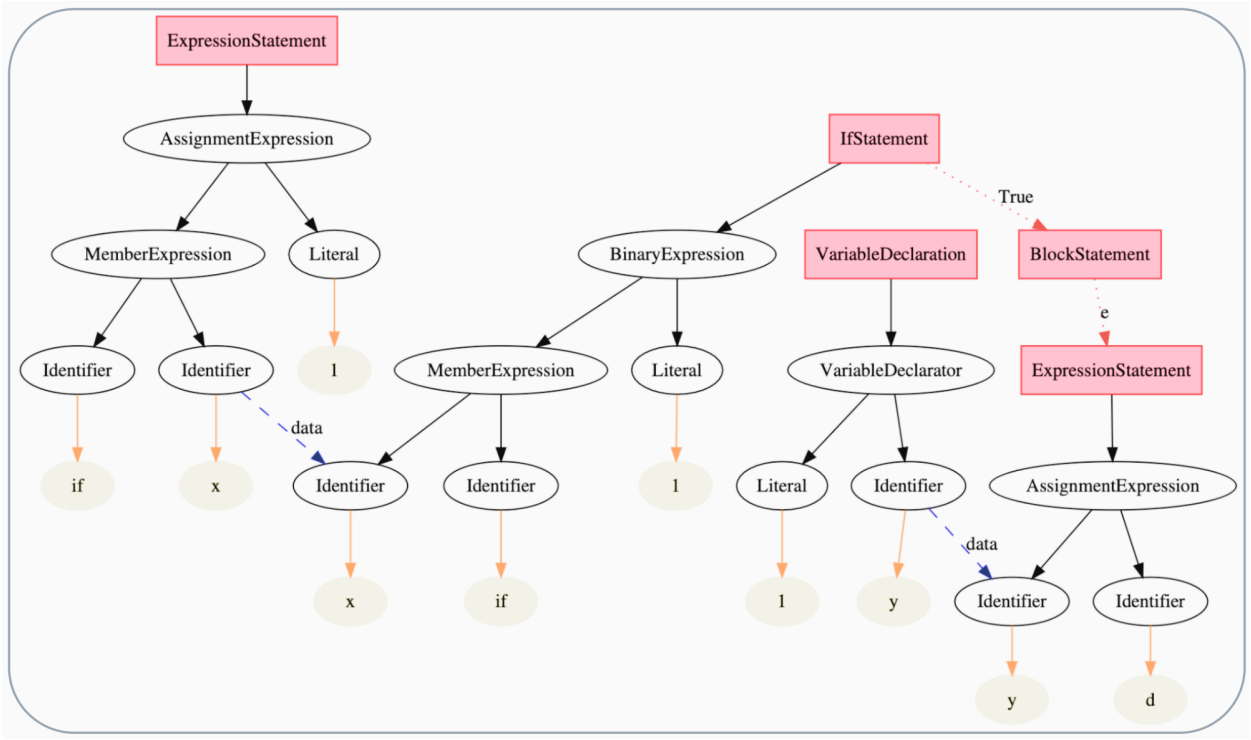
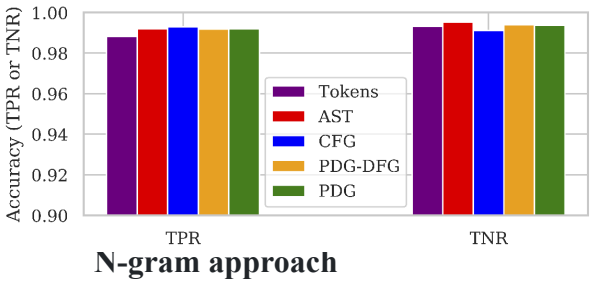
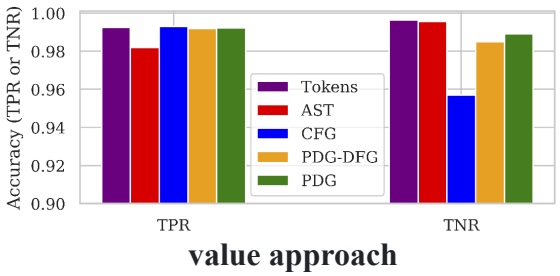


Figure 17: AST with Execution Path (data dependencies, data flow, number of features)

	Tokens	AST	CFG	PDG-DFG	PDG
ngrams	602	11,050	18,105	17,997	24,706
value	24,912	45,159	36,961	45,566	46,375

Moreover, the abstract code representation for the sample javascript code are considered in terms of n-gram features and to the corresponding node values as features. The n-gram features are used to identify specific patterns in javascript to improve detection accuracy, whilst the Node value features reflect the keyword value that refers to lexical or syntactic units. Analyzing the frequency of these features, malicious or benign javascript are identified accurately.





For learning and classification, a balanced collection set of both benign and malicious javascript files are provided to train the classifier such as, SVM, Naive Bayes and Random Forest. The performance of each classifier is assessed in terms of true-positive (accurate classification of a malicious script) and true- negative (correct classification of a benign script). To evaluate the accuracy, the composition of test sets (benign and malicious samples) are applied. The detection outcomes as AST based detection are superior to the other token based, CFG and PDG. AST based identification employs the combination of syntactic units while token based detection uses lexical values and the CFG, PDG further uses semantic informaiton. The essential points to consider from the above explanation, the lexical and syntactic units along with their frequency, offers insight into the properties of the code, that further aid to discover certain patterns suggesting it is malicious or benign.

PROPOSED SOLUTION

The javascript vulnerabilities and malware have advanced from simple to sophisticated with time. The scope of knowledge towards javascript malware requires a higher understanding of javascript language. With vulnerabilities, there are many attack vectors and the scope of the studies have narrowed down to one common type and the proposed solution in this research is aimed to address problems through learning from the real life issues and provide alternatives to the existing solutions.

A javascript malware detection and analysis tool (JS-MDA) is a web application that is built on the MERN stack with the integration of flask for python to support logich and machine learning algorithms that runs on the backend.MERN stack is a full stack web development technology. It comprises MongoDB, Express js, React js and Node js. The full stack development covers frontend , middleware and backend development. The frontend is handled by React application. It comprises reusable components, javascript XML, state and props, virtual DOM, event handler and API integration. The purpose of each concept lies to make UI structure more readable, promote code reuse, communication and data flow, data fetching, event handling, API integration to interact with backend servers via HTTP requests. Therefore, functions of each element from the UI such as buttons, textbox, search box etc is managed by react application to perform logical operation from the backend utilizing API (Application Programming Interface) and updates each changed state of components using virtual DOM (Document Object Manipulation)

Similarly, the backend uses Express.js a middleware pattern to handle requests and response with proper routing mechanism and Node.js a javascript runtime environment that uses event-driven architecture to allow I/O model for asynchronous programming. Each frontend logic such as search button or parsing, is handled and executed on the server side using node.js. Whereas a flask micro web framework for python programming works as an extension between web server and python applications. The API calls from the frontend can be redirected to the Flask endpoints and routes it to the designated functions from python applications. At the backend server the python applications like web scraping, trained machine learning model, identification

mechanism, parsing and static analysis are connected to the backend through flask. The frontend server and backend server runs on different ports to separately run and debug each part for development convvnience. CORS (Cross-Origin Resource Sharing) policy is implemented for interaction between resources running in different ports.

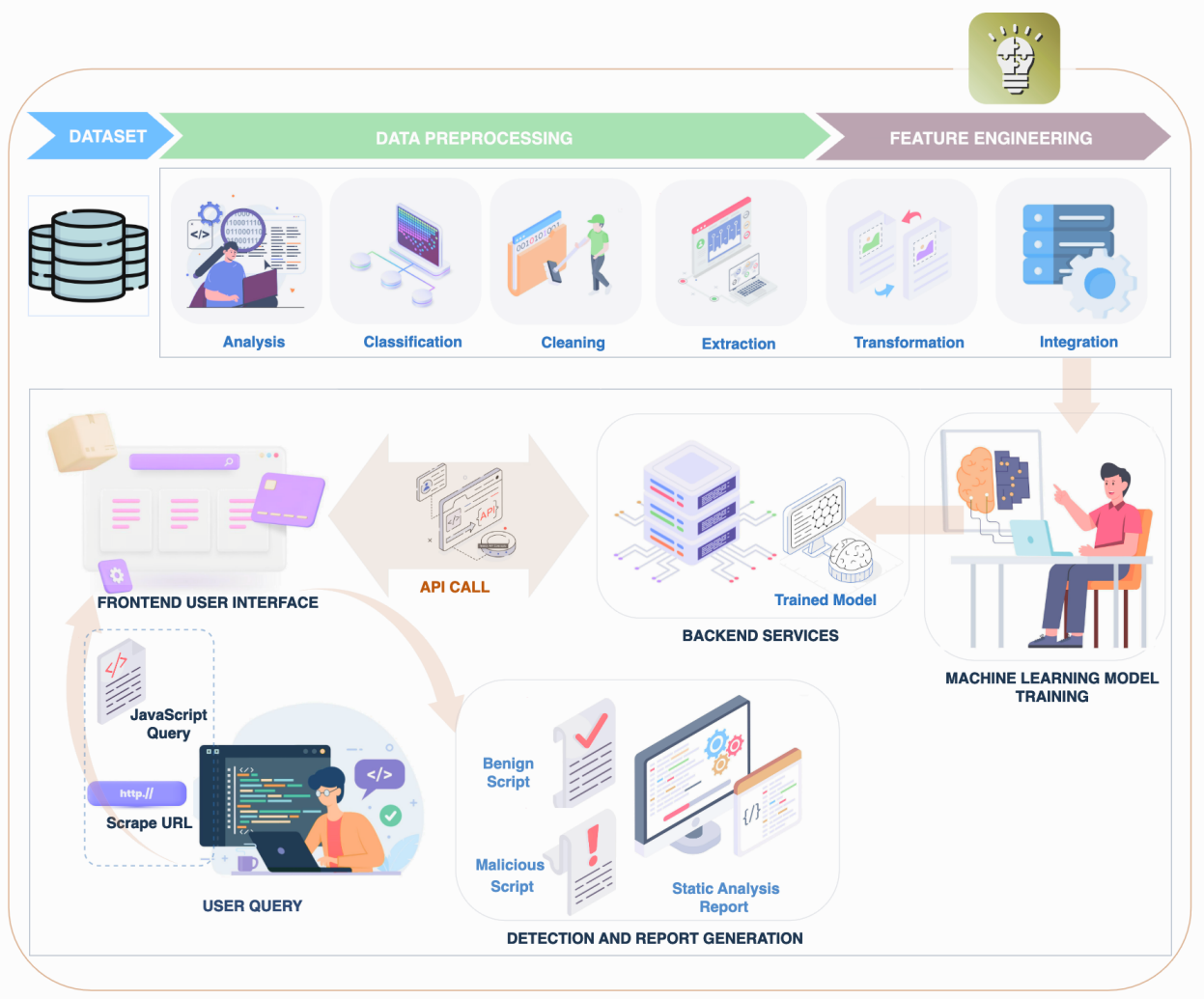


Figure 18: JS-MDA Working Mechanism

Moreover, machine learning models are trained separately with the collection of benign and malicious javascript datasets. The datasets are analysed, classified and transformed to fit for machine learning model training. The useful features and tokens are extracted to train the model and make decisions based on that. The trained model with higher accuracy rate is deployed in the backend to detect and identify javascript as malicious or benign.

Furthermore, the working mechanism of suggested alternative solution JS-MDA relies completely on the combined overall performance of the frontend, middleware, backend, the logics and machine learning algorithms attached to it. The design and development of JS-MDA is derived from deep study and analysis of working concepts that lie on the application CUJO,

JSTAP and ZOZZLE. All three tools depict different methods of detection and analysis on javascript malware, each method advancing from another and providing application in separate environmental setupups. JSTAP is a command line tool, CUJO is a web proxy and ZOZZLE is a browser application. However, each tool uses similar machine learning techniques for analysis on different types of data extraction leveraging detection mechanisms in its own way.

To this point, the working concept of JS-MDA is similar to those expressed above. The UI provides easy approachability for data extraction using url to scrape for only javascript tags. Whereas the built in function provides primary analysis of the script to identify morphism. Other functions to process javascript and extract features for analysis uses machine learning algorithms to identify the nature of the script as bening and malicious. Additional features are to generate static analysis report onthe javascript code where the result of the analysis clearly holds details on structure and intent of the code. The approach of JS-MDA provides solutions on the browser which may be useful for security analysts to do research on the malicious javascript and formulate mitigation strategies on site. Therefore, the implication of this web application helps to leverage the security posture of any organization, working closely behind the existing web infrastructure.

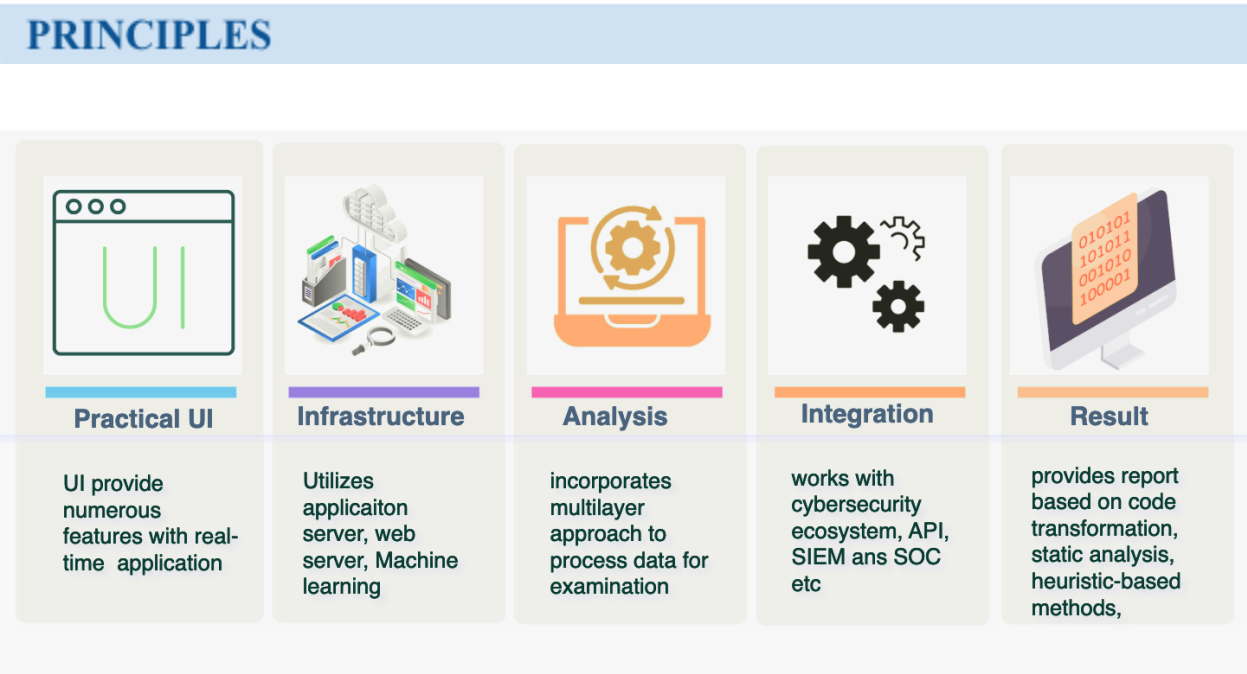


Figure 19: MERN stack Architecture

FUNCTIONALITY

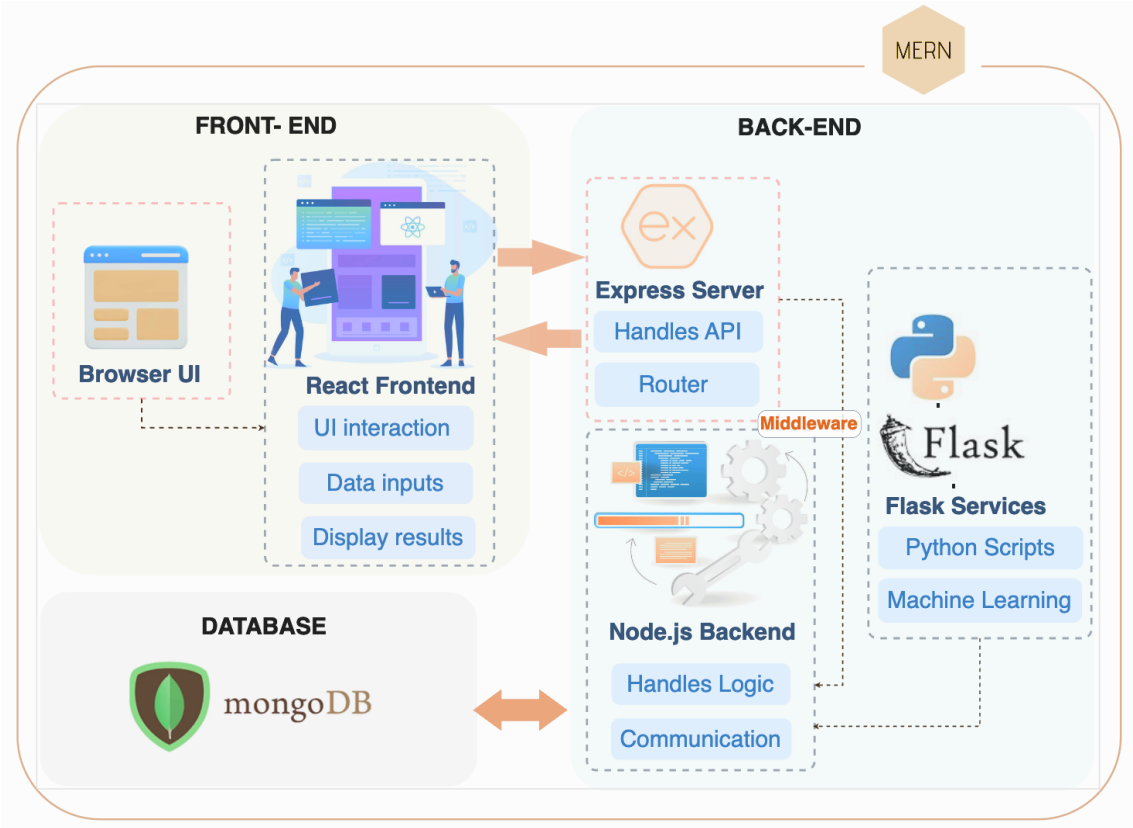


Figure 20: MERN stack Architecture

The development of the JS-MDA web application provides a powerful solution by combining the MERN stack (MongoDB, Express, React, and Node.js) with Flask services for executing Python scripts and machine learning algorithms. This architecture not only offers the efficiency and necessary capabilities for precise detection and analysis of malicious JavaScript, but it also offers an intuitive user experience. By using Flask services, the application gains a substantial amount of computational capability.

The React.js frontend, as the first component of the architecture, enables and simplifies user engagement. Express.js acts as the middleware to accept these requests and then routes them to the appropriate backend services. Node.js in the backend is responsible for the primary logic.

There are several advantages to integrating Flask services with the MERN stack. Flask greatly boosts processing capacity, making it feasible to execute machine learning algorithms and complicated Python scripts. Utilizing the capabilities of both Python and JavaScript, the application delivers a robust malware detection and analysis solution. (Erickson, 2024)

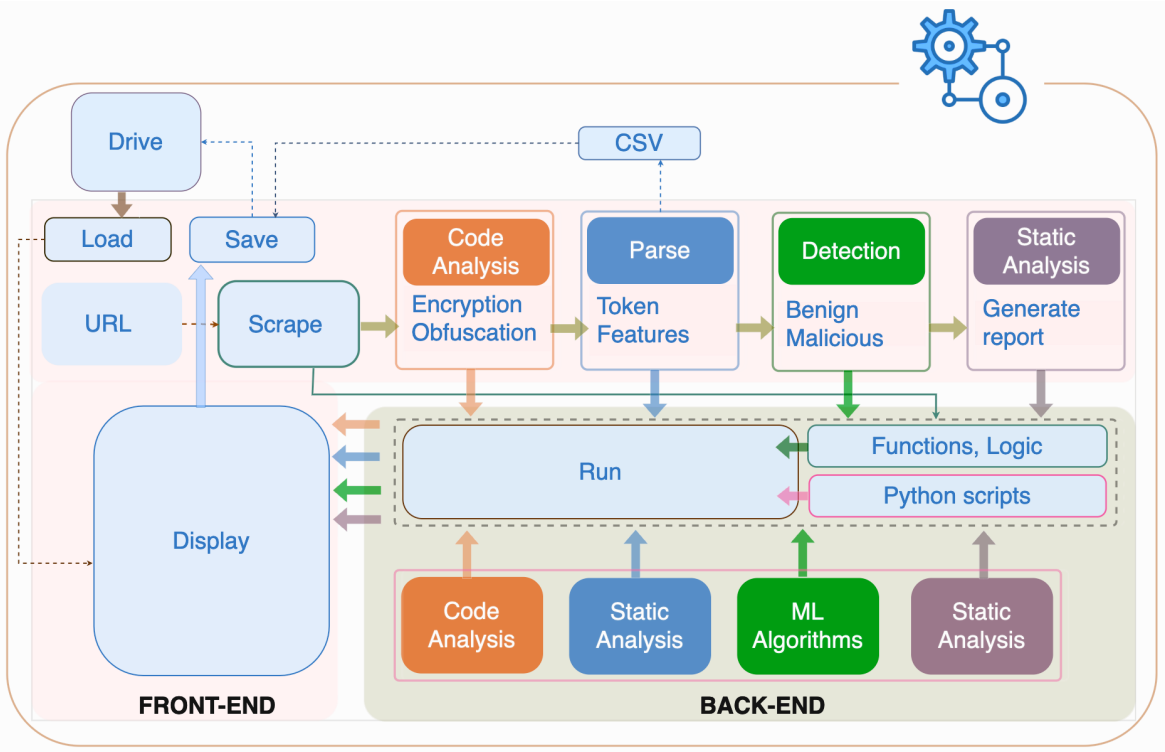


Figure 21: JS-MDA Web app Functionality Diagram

The process begins with the user interaction enabled by the frontend of the application constructed using React. Users may utilize the search box to input URLs for scraping static and dynamic web pages. The backend, driven by Node.js, conducts the first processing of these sepcified URL requests, commences the web scraping process using Python libraries such as BeautifulSoup and parsing HTML, and extracts any JavaScript files linked inside the HTML text, including both inline scripts and external script files. These received JavaScripts are sent using API to display it in the textbox, it is also stored temporarily in the backend. It may be further stored locally or instantly forwarded for code analysis. Furthermore, the user can also upload a javascript file or enter javascript code directly into the text area. This adaptability implies that users may study javascripts either by scraping from sites or loading scripts from local systems, laying the foundation for processing and analysis.

For code analysis, the frontend communicates the file contents to the backend via an HTTP request. The contents are read using the FileReader (MozDevNet, 2024) API on the frontend and are passed down to the backend for processing. The backend holds the logic to analyze and detect the encryption and obfuscation in the code, and the result of the analysis is displayed in the textbox. If any concealment is found in the code, the user has to manually decode it using other online tools.(encode, 2010)

Further, the parsing process includes examining the JavaScript code to extract essential elements. The backend employs the Esprima library, a JavaScript parser, to tokenize each small componentes of the code based on lexical and syntactic features. The result of parsing is

automatically saved in CSV format locally. It can further be loaded into the textbox for data analysis purposes.

The detection process primarily includes feature extraction, which discovers and measures certain traits important for identification (Agarwal, 2024). These criteria may include the occurrence of patterns of tokens and features. The result of the parser saved as CSV must be loaded, and the content must be processed to retrieve useful characteristics and then encoded into a structured data format to permit further analysis by machine learning algorithms. The application's main feature is the categorization of JavaScript code as either malicious or benign. The pre-trained machine learning model is used for the classification of JavaScript. The models are taught on a large dataset of benign and malicious JavaScript samples. The algorithms used by the machine learning model can be decision trees, random forests, and support vector machines (Smolyakov, 2023). The structured data serve as input variables for the model, which further analyzes it and provides a classification result of javascript code as malicious or benign.

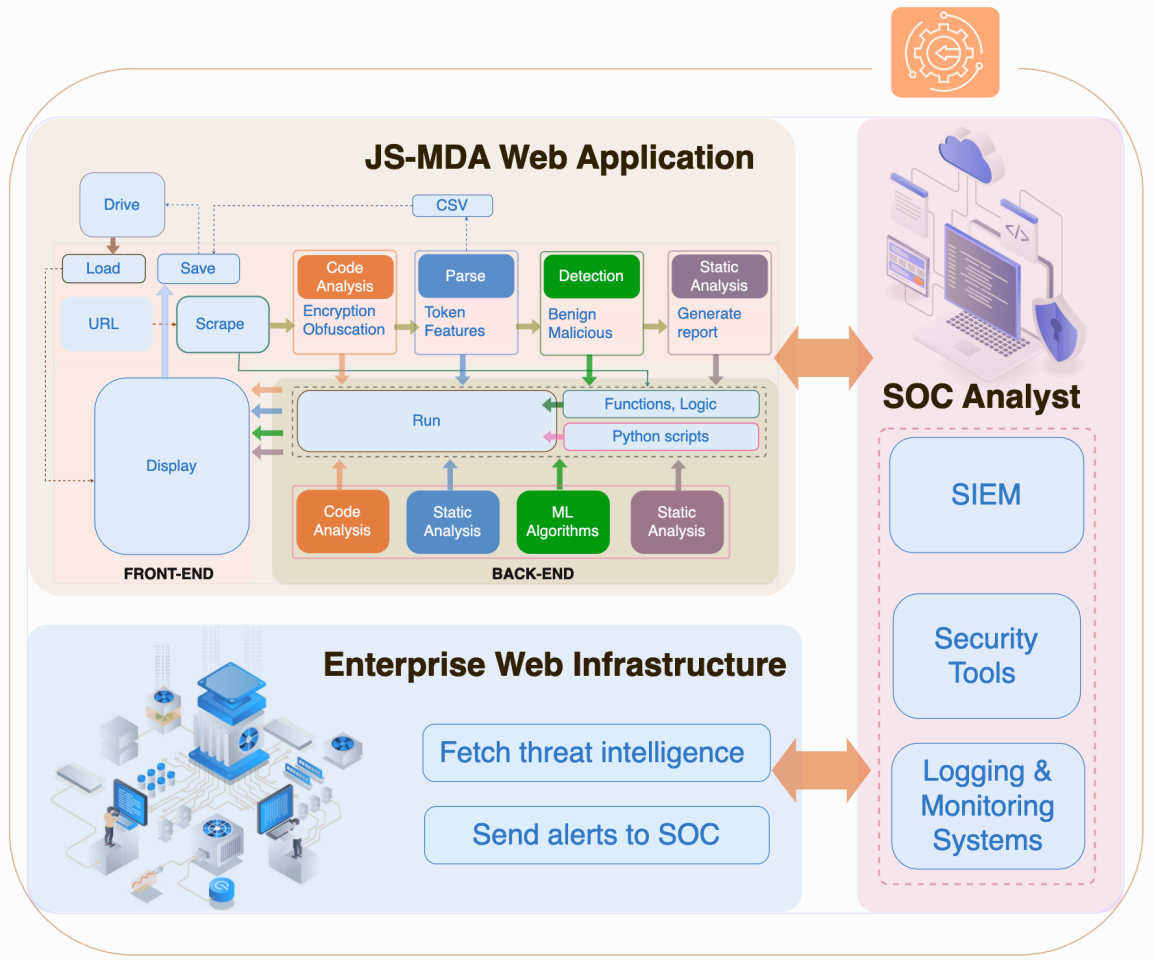


Figure 22: Integration of Complete Cybersecurity Framework with JS-MDA app

Finally, the static analysis generates a report based on the parsed JavaScript code. It provides detailed information on the code's structure and any patterns discovered. The report may contain information about the number of functions, loops, usage of eval functions, and characteristics that may indicate malicious intent. The report is structured in text format to pass clear and simple information to the user. This report delivers significant information on the JavaScript code, which can be further used by malware analysts for the precaution and mitigation of any vulnerability caused by malicious JavaScript code. (Hughes, 2021)

Integrating the JS-MDA web application with the enterprise web infrastructure is a complex and critical task that ensures the protection of organizational resources. A simple overview of the integration is depicted in the figure, which visualizes each block as a whole and doesn't go much deeper, as it is beyond the scope of the research paper. Furthermore, the overview entails connecting the web application to the SOC analyst and the enterprise web infrastructure. The main purpose is to show the possibility of an ecosystem where detection, monitoring, and response processes are streamlined to enhance the overall security posture of the organization.

Furthermore, JS-MDA communicates with the EWI systems, mainly network security infrastructure such as firewalls, intrusion detection and prevention systems, and Web Application Firewalls (WAFs), to identify threats. When JS-MDA detects any suspicious script, the alert is sent to the SOC analyst via these security systems. Based on the type of malicious scripts, EWI systems are updated and configured to monitor, block, and protect web applications. These security measures work together to provide a layer of defense to protect against numerous attacks.(Karaarslan, Tuglular, Sengonca, 2007)

More importantly, the logging and monitoring systems provide secure access to the JS-MDA web app and the essential SOC tools. It is a centralized management system for user credentials and access permissions, making it simple to enforce security policies and manage user roles. whereas SIEM integration tracks logs and events to identify potential threats and prioritize incidents based on their severity. The SOC analyst uses this information to monitor the security status, investigate incidents, and coordinate response efforts.(Cyberpedia, 2024)

Overall, JS-MDA, together with the Security Operations Center and the enterprise web infrastructure, leverages the organization's defense position and provides an adaptable approach that can evolve with growing new threats and changing legal frameworks.

EXPERIMENT AND RESULTS

The JS-MDA application is built on the MERN stack and uses Flask to run Python scripts. The experiment is performed on each element in the stack. The primary purpose is to ensure simple integration between the frontend and backend. Also, provide a user-friendly interface for carrying out analysis steps and displaying requested results. Furthermore, the outcomes of

functions such as scraping, parsing, code analysis, and producing reports were displayed in real time. Therefore, to confirm the efficiency of the JS-MDA app,

Furthermore, the React frontend's connectivity with the Express/Node.js backend is verified for data transmission and reception. The RESTful API endpoints that handled the communication also gave instant feedback on the process's status. The status was displayed on the front end.
The experiment includes communication between the Node.js backend and a Flask server. It runs Python scripts that provide true functionality to the frontend buttons. The goal was to minimize latency when sending requests from the front end to the backend. This included triggering Python scripts, saving results, and displaying feedback. JSON and CSV files are being used to communicate data from the backend to the Flask server.

Moreover, the machine learning classifiers Random Forest, Support Vector Machine, and Logistic Regression are trained on either benign or malicious JavaScript files. Tokenization and feature list mapping are done manually using the Esprima parser. The training and testing datasets are split 80:20. The dataset samples are balanced by SMOTE and PCA. This strategy improved the models by lowering training times and overfitting. The analysis of misclassified samples indicated that certain malicious scripts were mistakenly classified as benign due to subtle traits that were not captured successfully during feature extraction. These misclassifications were connected with attributes with low significance ratings.

Furthermore, the models were assessed using metrics such as accuracy, precision, recall, F1-score, and AUC-ROC. The evaluation results showed that the Random Forest model outperformed SVM and LR in terms of accuracy and recall. Thus, the RF model is employed to detect JavaScript samples.

The trials indicated that each function of the JS-MDA program performed efficiently in identifying JavaScript files as malicious or benign. However, the effectiveness of ML detection is dependent on the quality of the javascript data to be analyzed. The scraping and parsing procedure must produce accurate data so that the detection engine can make predictions. Furthermore, other aspects of the application functioned properly. It demonstrates the successful integration of the MERN stack and ML as a whole. These favorable results of JS-MDA allow for its use in EWI with SOC.

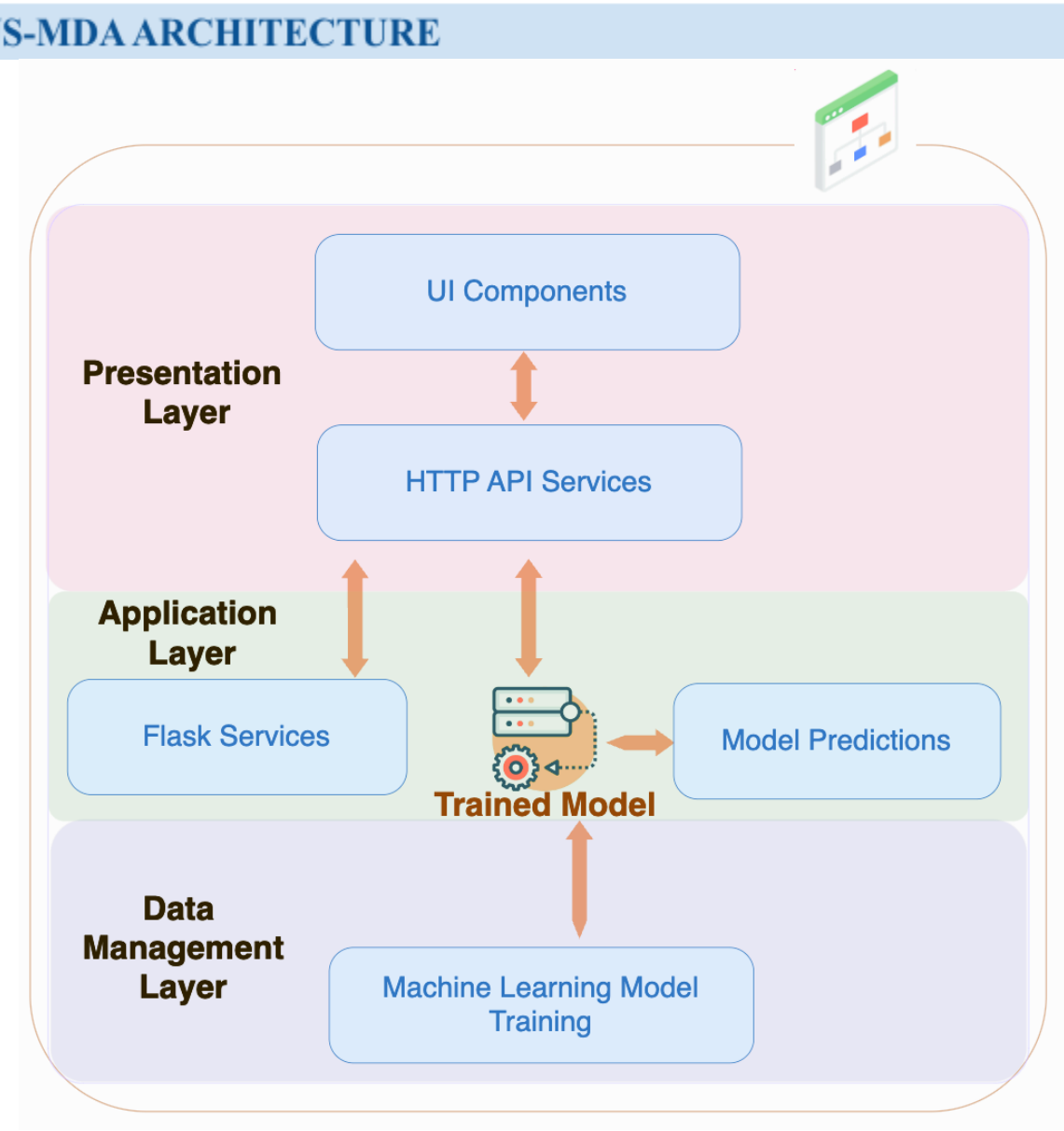


Figure 23: JS-MDA Architecture

The JS-MDA web application works with a three-tier architecture: the presentation layer, the application layer, and the data management layer. The client layer collects user-defined input and displays the results appropriately. The application layer processes the inputs (such as JavaScript) and employs multiple levels of filtering in the script to prepare the pre-trained model for classification. The data management layer holds the model and maintains temporary data. Using the pre-trained model, the following configuration allows the identification of javascript code as malicious or benign.

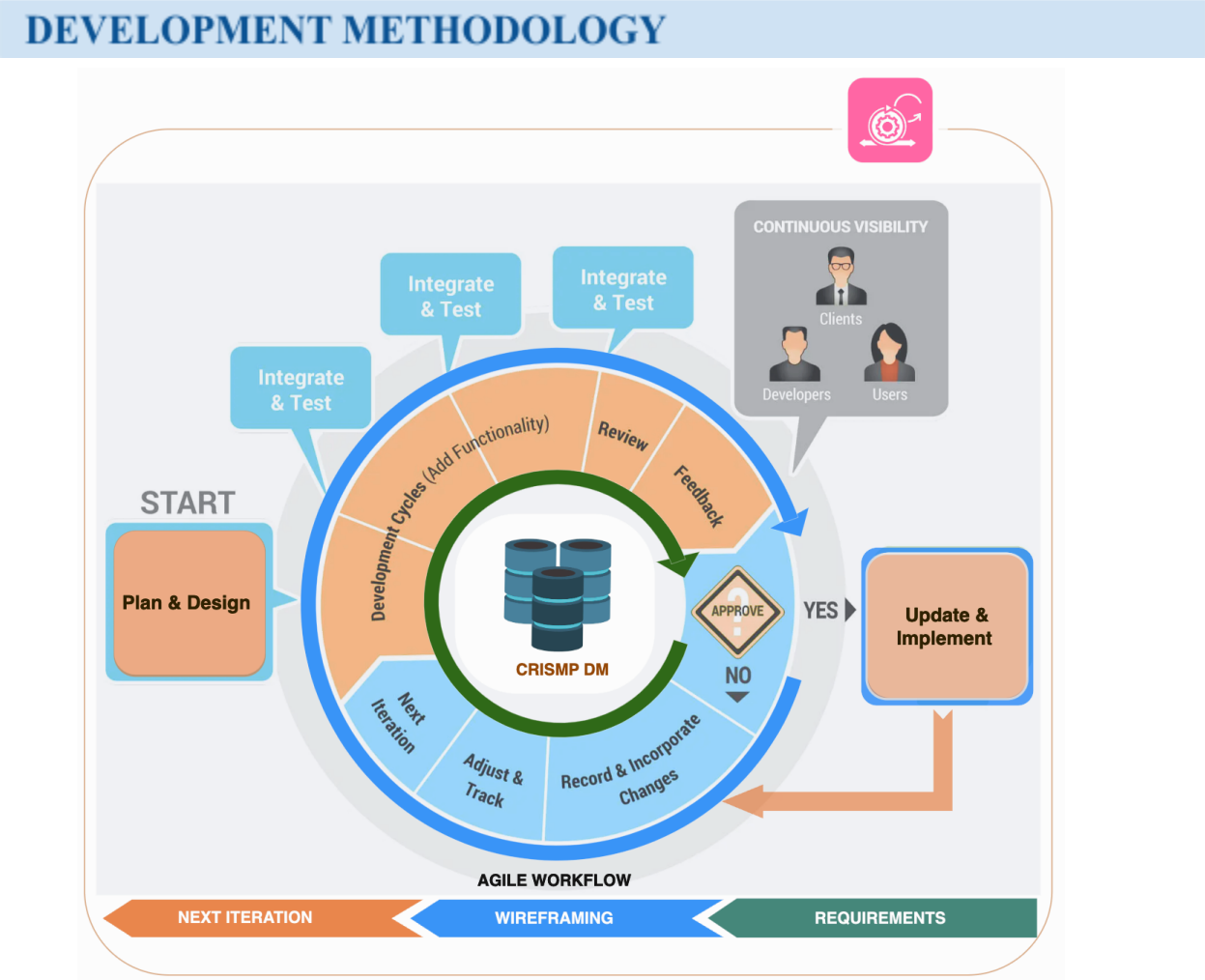


Figure 24: Combined Agile and CRISP-DM approach

The combination of Agile methodology with CRISP-DM (Cross-Industry Standard Process for Data Mining) offers a rigorous framework for building and developing a model. The agile application development technique aims to streamline the process of developing an application plan by breaking down complex requirements into small, individual blocks. A small block of activities can be analyzed to make more predictions about the amount of work needed to accommodate the possible changes. Moreover, CRISP-DM's structured approach to data collection encompasses business knowledge, data understanding, data preparation, modeling, assessment, and deployment. The Agile methodology places a significant emphasis on stakeholder identification and their active participation throughout the entire application development process. The CRISP-DM stages the model's performance, and the deployment step, where the model is placed into production, guarantees that any modifications necessary in the model or the data it acts on can be promptly included, keeping the project's development relevance and pace (Edeki, *Agile Software Development methodology* 2015). Finally, the combination gives a complete foundation for creating quality, relevant, and efficient detection models. The design approach stresses the necessity of Agile's iterative, flexible approach. It

offers a dynamic and responsive environment for model development. It further helps to build an adaptive and effective detection-making model to provide a solution to the existing problem. A solution like the JS-MDA web application delivers real value and insights, enabling improved decision-making as a competitive advantage over traditional detection models.

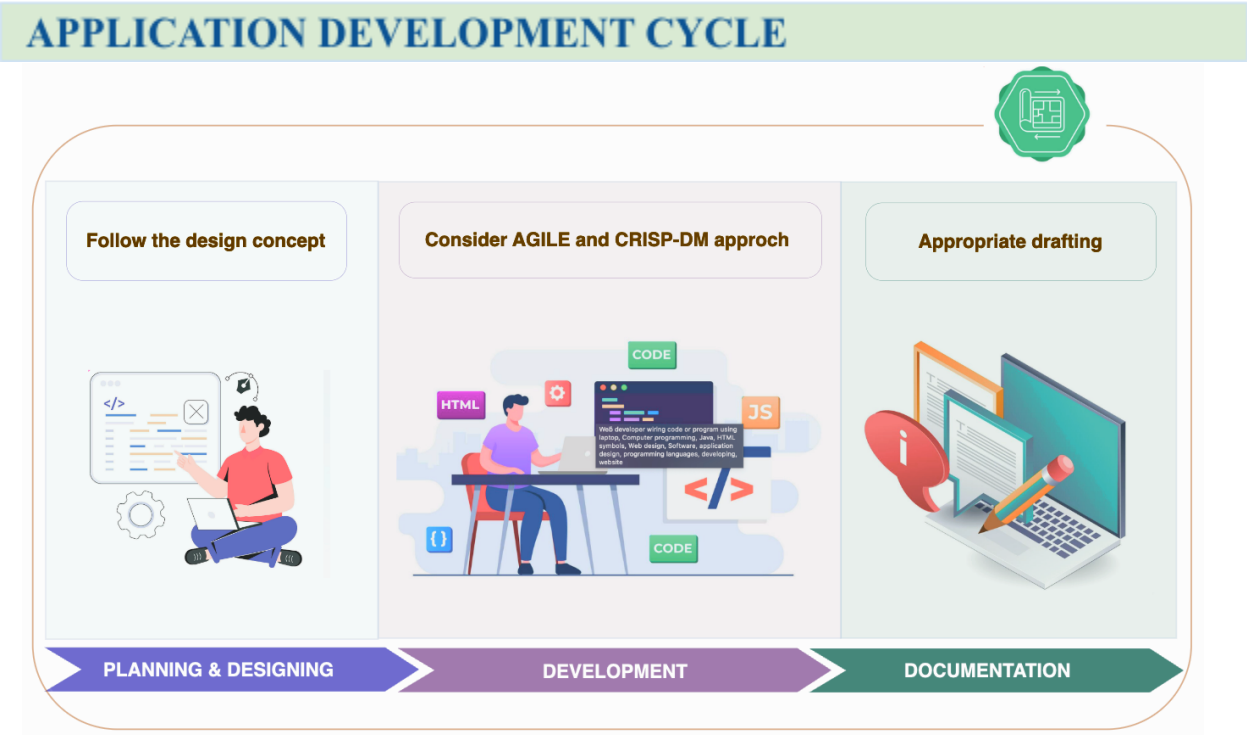


Figure 25: Application Development Cycle

The application development cycle is a systematic process that provides an achievable approach to managing the different stages of app development phases. It consists of planning and designing, development, and documentation. The planning and designing step is an introductory step that defines the project's scope, goals, and requirements and implements the design elements accordingly. The development phase is the actual creation of the application system. This phase encompasses continuous development, testing, integration, and troubleshooting. It is an iterative cycle of coding and testing. The building of essential infrastructure, such as servers and databases, needed to support the application is done in this phase, which adheres to the criteria given in the planning phase.Finally, the documentation phase is important for maintaining thorough records of the whole development process. Documentation is a technical handbook that allows users to assist with maintenance instructions. It provides comprehensive knowledge of the system's design, functionality, and operational processes to serve as a reference for identifying, upgrading, and improving the program.*Application development life cycle: An overview* 2010)

PLANNING & DESIGNING

The planning stage is a primary step in establishing the application's complete framework. Proper timelines, and potential risks related to the development of the project are identified beforehand with the mitigation strategies. Moreover, feasibility studies and requirement analyses are conducted to understand the technical requirements of the project. The project's goals and expectations are clear at this stage. Furthermore, the design must incorporate the needs identified during the planning process into a single design concept. This contains the system architecture and comprehensive design elements. The system architecture provides the program's overall framework. It comprises hardware and software components, data flow, and communication interfaces. The complete design elements cover user interface designs, data communication formats, and particular detection algorithms.

DEVELOPMENT

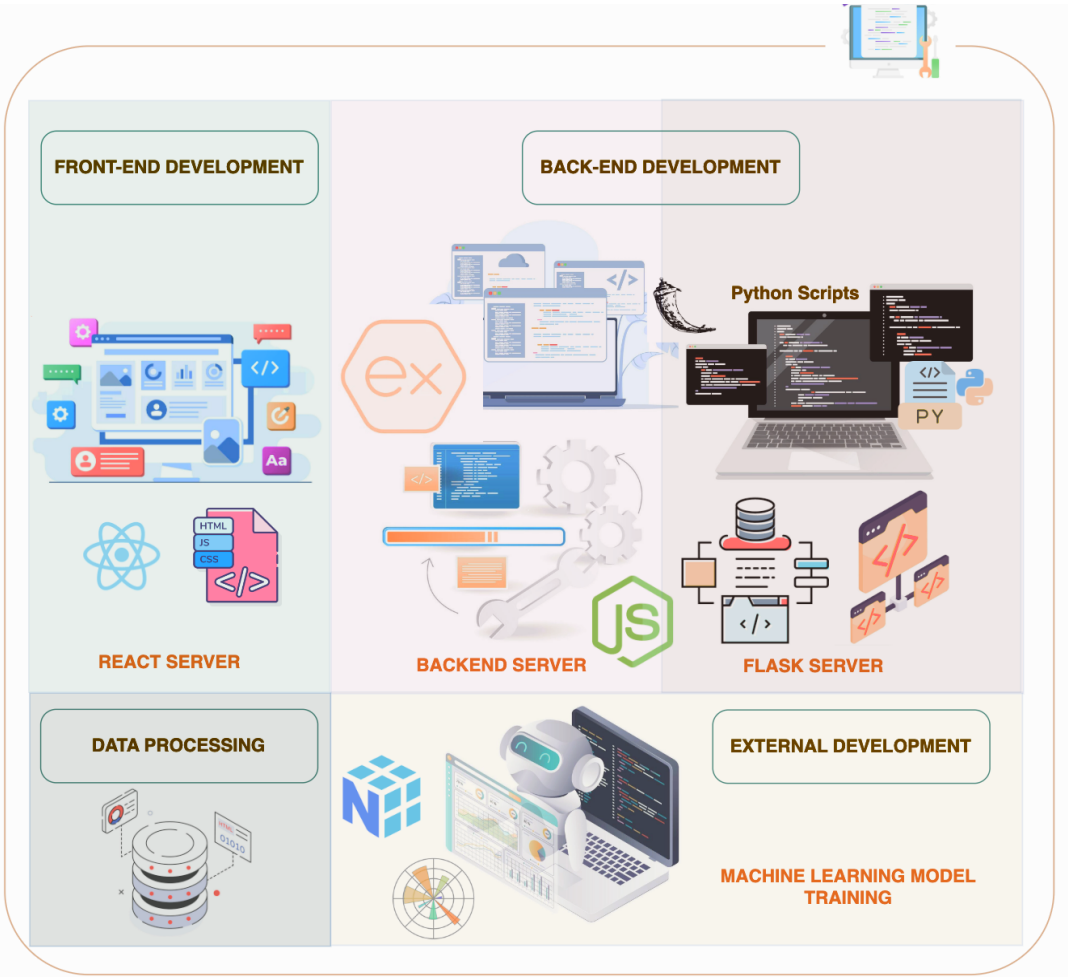


Figure 26: Infrastructure Development

Web infrastructure development refers to the design and arrangement of the components and technology essential to enabling the functioning of online applications. It is a process that

comprises setting up and maintaining multiple components to enable an effective, secure, and scalable online application. The frontend server provides the user interface, while the backend server performs business logic and transactions. Flask servers and other microservices are able to provide specific functionality and methods to ensure an effective detection mechanism. External integrations provide access to third-party services, such as integrating machine learning models into the backend systems. Together, these components build an agile web infrastructure that can host JS-MDA applications and related services.

FRONTEND

React-based JavaScript-MDA web application servers rely on environment variables, dependency management, and component design. Its interaction with the backend is intended to enable the execution of the main functionality in the JavaScript-MDA application scope. Environment variables are defined in the.env file and may be accessed throughout the program. The dependencies refer to the external libraries and tools that are essential for the application to run, and the components are the building blocks of a web application that include functional and class components. The components are used to define the UI elements and logic, state management, routing components for navigation, and HTTP clients like Axios for communication with the backend. (Minnick, 2022)

ReactDOM is used to set up the React application when rendering components to the DOM. ReactDOM.render helps React control the UI by initializing the primary app components into the DOM element. On a basic level, the program runs under app.js. It combines file upload, code analysis, static analysis and detection methods, and UrlInput. One important file handling the project's dependencies and scripts is package.json. It offers a constant development environment across several configurations and accurate installation of all dependencies. To maintain and control the state It employs a state hook.

HTTP inquiries are sent to the backend, and replies are received via Axios. PORT=3000 defines the port indicated on the environment variables where the React development server will operate. The frontend component links itself to the backend using specified API paths. These paths match different purposes, including scraping, file upload, code analysis, parsing, static analysis, and script detection utilizing machine learning. The backend, which runs Node.js and Express, handles these requests, runs the required logic, produces outputs, and sends them to the frontend using an API.

BACKEND

The backend of a JS-MDA web application plays a crucial role in ensuring the system's performance. The server is built using Node.js and Express.js. It is combined with machine learning models and other components to provide additional functions with Flask. It focuses on managing server-side logic, handling data processing, and providing frontend functions. It employs technologies like Node.js, Express.js, and Flask to execute external Python scripts.

Node.js is a core backend that serves as the runtime environment. It allows for efficient query processing. Moreover, Express.js is used to develop RESTful APIs and streamlines the process of routing requests, managing middleware, and connecting with other components, making it perfect for building server-side logic.

The fundamental configuration of a server script is in a file like server.js; it works as the basis for the construction of the backend to carry out numerous functions. The code can handle HTTP requests and create an Express application. Configures a port specified by an environment variable, which defaults to 5000. Uses express.js to construct APIs that manage routes and handle various sorts of HTTP requests. The router file specifies particular routes and logic, which makes the application simple and organized.

Trained Machine learning models and other relevant Python scripts are incorporated, utilizing RESTful APIs. When Flask is combined with a Node.js backend, the Flask application runs on a different server. Node.js connects with Flask endpoints using an API request, which is conducted using Axios.(Brown, 2014)

EXTERNAL DEVELOPMENT

The CRISP-DM (Cross-Industry Standard Process for Data Mining) approach is utilized for the development and implementation of a machine learning (ML) model for the JS-MDA web-based application. It is an approachable method for data collection and machine learning training models. This technique offers direction for the entire machine learning process, which covers data handling, categorization, feature extraction, transformation, training, and evaluation.

Furthermore, the first step of the CRISP-DM cycle focuses on setting up the ML model's goals and requirements for malicious Javascript detection. Knowledge drives the phases of the CRISP-DM cycle, which include all activities aimed at improving accurate detection capabilities.

Moreover, Python packages like Pandas are utilized to find insights about data allocation, feature properties, and any possible data quality concerns that require attention. At this stage, Python's NumPy module assists in processing numerical data and executing mathematical operations. Scikit-learn includes a broad variety of classification methods and processing tools that incorporate support vector machines (SVM), random forests, and gradient boosting. The selected algorithms are trained on the labeled dataset.

The aim is to train a model that can properly identify new JavaScript files as either malicious or benign. Using Scikit-learn’s evaluation algorithms, various metrics covering accuracy, precision, recall, and F1 score are determined. Matplotlib is used to illustrate the model’s performance using plots and charts, such as ROC curves or precision-recall curves. Each step highlights the areas for improvement, assesses the model based on performance, and makes iterative modifications to the features, methods, and parameters for the desired detection accuracy. (Grigorev, 2021)

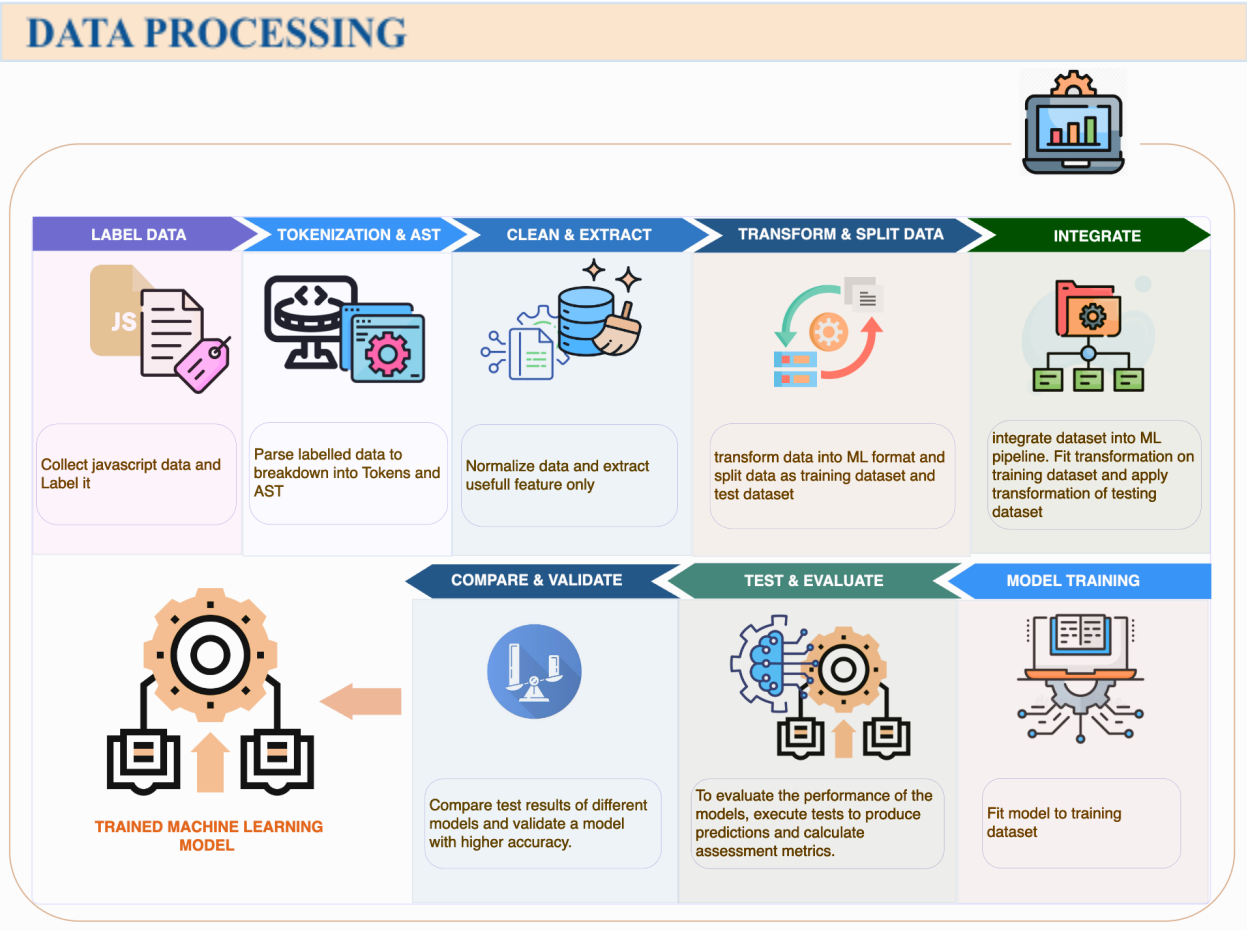


Figure 27: Machine Learning Data Processing

The first phase of data processing is the acquisition of raw data, which usually entails collecting datasets including both harmful and benign information from various sources, such as data warehouse websites, repositories of clean code, and databases of malicious scripts. Web scraping is often necessary when the desired data is not easily accessible. Only reliable data sources are used, adhering to legal and ethical standards. It ensures the data's quality and integrity are preserved. The dataset is carefully annotated with labels indicating whether it is benign or malicious. Moreover data analysis helps with the visualization and interpretation of the datasets. The example data representation histogram is below. (Singh, 2020),

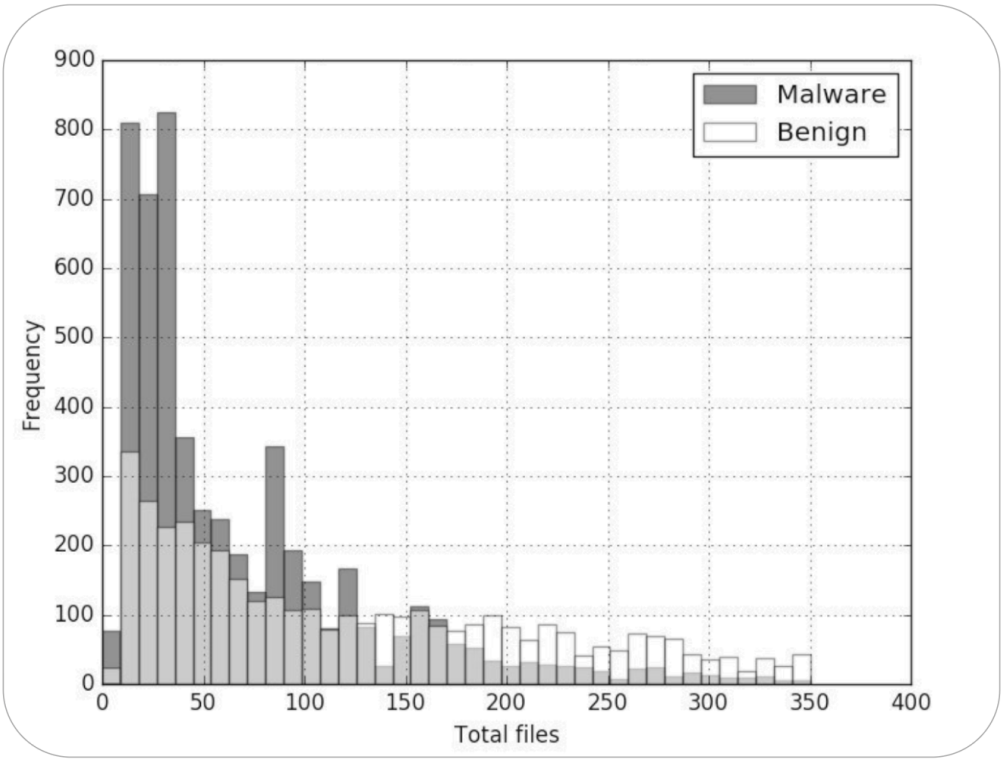


Figure 28: Data Analysis

[(Kumar, 2017)]

```
// Function to steal cookies
function stealCookies() {
  var cookies = document.cookie;
  // Send the cookies to the attacker's server
  var xhr = new XMLHttpRequest();
  xhr.open("POST", "http://malicious-site.com/steal-cookies", true);
  xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
  xhr.send("cookies=" + encodeURIComponent(cookies));
}
// Function to redirect to a malicious site
function redirectToMaliciousSite() {
  window.location = "http://malicious-site.com";
}
// Trigger the malicious actions
stealCookies();
redirectToMaliciousSite();
```

Malicious Javascript Code

```
// Function to add two numbers
function addNumbers(a, b) {
  return a + b;
}
// Display the result in the console
console.log(addNumbers(5, 10)); // Expected output: 15

// Function to change the background color of a page
function changeBackgroundColor(color) {
  document.body.style.backgroundColor = color;
}
// Change the background color to light blue
changeBackgroundColor("lightblue");
```

Benign Javascript Code

Figure 29: Javascript Dataset Example

Moreover, tokenization is the process of dividing the original JavaScript code into more manageable components. The manual categorization of tokens and features is used to classify the context of JavaScript. Tokens include keywords, operators, identifiers, literals, and more, while features include eval, document.write, and others. Tokenization example is illustrated below.

Sequence	Token	Type
1	function	Keyword
2	stealCookies	Identifier
3	(	Punctuation
6	document.cookie	Property
7	=	Operator
8	"username="	String
9	+	Operator
10	encodeURIComponent	Function Identifier
11	(	Punctuation
12	document.cookie	Property
20	"POST"	String
21	,	Punctuation
22	"http://malicious.com/cookies"	String

Figure 30: Illustration of Tokenizaiton Process.

After tokenization, the next phase involves cleaning the data and extracting features. Cleaning entails deleting unnecessary information. This could contain uninformative tokens, such as standard library functions or common variable names. Whereas Feature extraction is the process of extracting relevant patterns from JavaScript code. Analyzing the incidence of keywords, such as nested loops or recursive functions, the employment of eval() or setTimeout() procedures, etc. These sorts of characteristics are in forms that can be employed and interpreted by machine learning algorithms.

Addition to, the characteristics then need to be converted into a format suitable for model training. Normalization transforms categorical variables into numerical representations. Then the dataset is often divided into a 80:20 ratio as training and testing sets. It is vital to guarantee that the model has enough data to learn from and a sufficient quantity of unseen data for a meaningful assessment after training.

Further integration entails integrating two datasets into a uniform format, ensuring that all the data follows the same structure and has the same properties. Consistency throughout the dataset is critical for proper training and assessment. This phase is important to balancing benign and harmful samples. Other approaches to balancing the dataset might include oversampling, undersampling, or synthetic data synthesis.

The model training method entails putting the training data into a selected algorithm and enabling it to understand the patterns of benign and malicious scripts. The model iteratively adjusts and trains for predictions on real labels, resulting in a prediction model that is capable of reliably categorizing new and unseen javascript data.

After training, the model is evaluated using Key metrics such as accuracy, precision, recall, and F1 score. The model's performance shows false positives and false negatives rates. The testing technique involves models being trained and tested multiple times, each time with a different subset as the testing set to ensure that the model's performance remains consistent over new datasets..

To benchmark the trained model against other models comparison and validation is done assessing its relative performance. This phase determines the more effective algorithm and ensures it meets the objectives and equirements. In case of JS-MDA it must prioritize the model which has less false negatives and results in a few more false positives. The validation phase checks the reliability of the model.

By following the CRISP-DM cycle, this process is structured and systematic, ensuring that each stage is aligned with the ultimate goal of accurately identifying malicious scripts. This approach not only enhances the reliability of the model but also provides a framework for continuous improvement and adaptation to new challenges in the cybersecurity domain.

CONTINUOUS ITERATION AND DEPLOYMENT CYCLE

Application testing involves carefully analyzing the JS-MDA web application by running it in a controlled environment and validating that it operates as intended. It is an important stage in the software development lifecycle. The testing of essential features and functions is examined to determine how the application functions together to offer effective detection and analysis results. Moreover, the different components of the program, such as procedures that parse and tokenize JavaScript code, code analysis, and interaction with the trained machine learning models, confirm that each component performs appropriately on its own.In addition, integration testing confirms that the services function together in the form of interaction between the frontend React components and the backend Express.js APIs, adjusting the data flows accordingly.(design and develop, 2021)

Functional testing involves feeding both benign and malicious scripts to the system and confirming that it properly identifies them. Moreover, the application must operate under different scenarios to make certain that the system can manage heavy traffic, analyze massive datasets, and reply rapidly. Security testing means looking for weaknesses in order to assure secure data handling methods.

Once testing is complete and the critical input from the application is taken, the feedback assists in detecting difficulties, analyzing the user experience, and collecting recommendations for

changes. As a developer of the program, the technical input for introducing features, the performance of the code, and prospective areas of improvement techniques are set.(Bhardwaj, 2024)

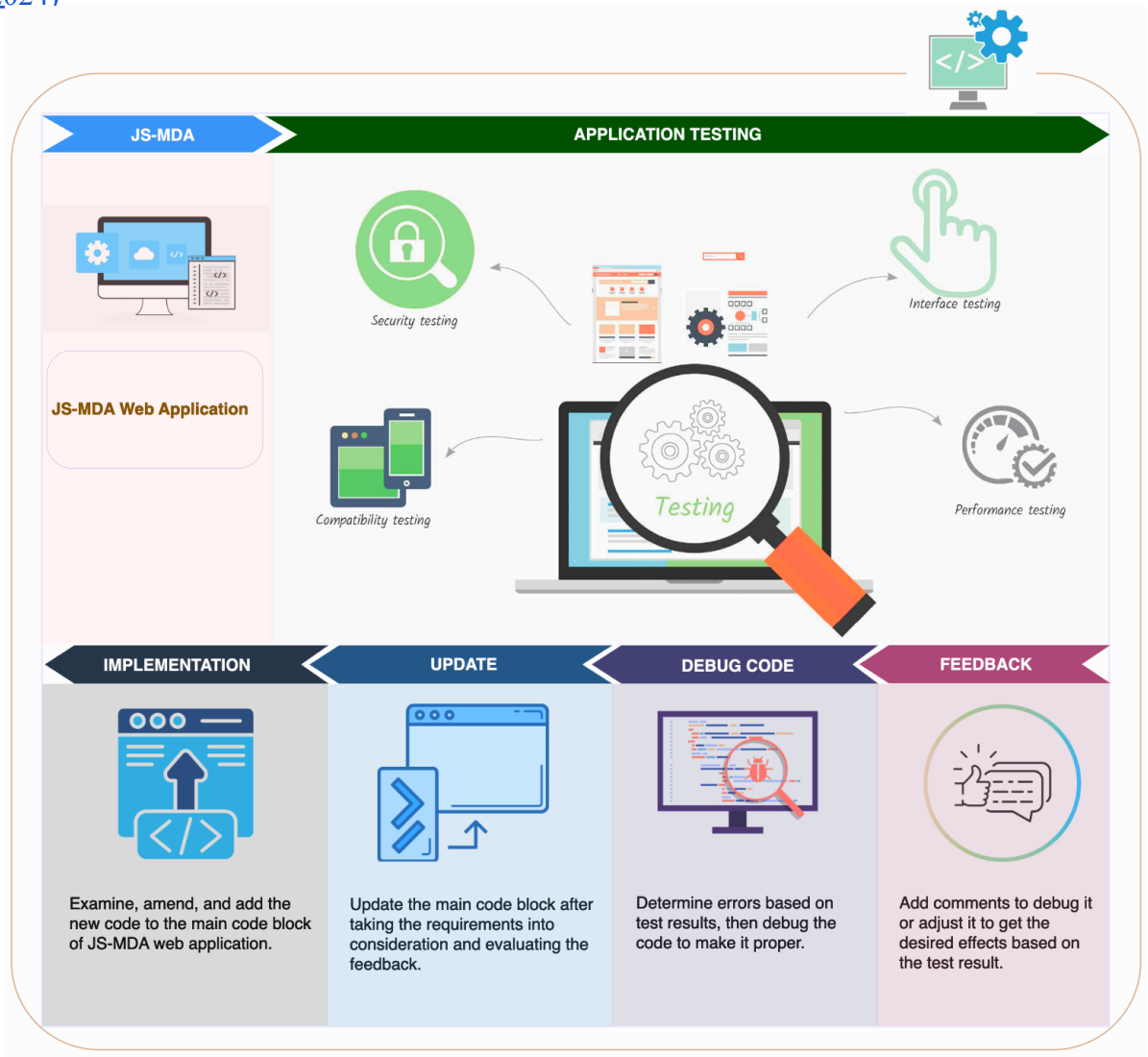


Figure 31: Continuous Iteration and Development Cycle

All of the input is helpful in the debugging process to correct program faults. Debugging frequently entails analyzing logs, debugging code, and utilizing diagnostic tools. For instance, if a code analysis problem was detected, the appropriate code portion would be changed, re-tested, and checked to acknowledge that the issues had been fixed.

The JS-MDA application must adapt to new needs and stay up-to-date with new threats. It entails extending current functionality and the ability to evaluate complex JavaScripts using machine learning models built on fresh datasets.

The improvements are tested and confirmed, then merged into the main code. It is carefully controlled for an uninterrupted change. Continuous monitoring of the new systems is necessary so that they work as planned. Evaluation of feedback, system performance, and the accuracy of

javascript detection are immediately handled to make sure the system stays stable and runs effectively.(Kate Eby, 2019)

DOCUMENTATION

The major objective of documentation is to offer clear and accurate information to users, and security analysts. It ensures a common understanding of the system's architecture, operating processes, and the logic behind design decisions in the context of a JS-MDA web application program. It helps resolve difficulties, improve performance, and verify compliance with security requirements. Furthermore, detailed documentation facilitates knowledge transfer and increases the likelihood of receiving concise support from other experts. Visual aids such as diagrams, flowcharts, and tables help explain the complicated processes. Regular updates to the documentation are necessary to ensure it stays correct and current, particularly in agile development settings where changes are frequent.(AltexSoft, 2024)

In the constant development and maintenance of a web application, documentation plays a vital role in keeping records that grow with the system, preserving the history of modifications, rationales behind choices, and future growth plans. During debugging and troubleshooting, developers may resort to the documentation to understand the system’s architecture and operating logic, which assists in detecting and resolving problems rapidly.(Javed, 2016)

Documentation also assists the machine learning model's ongoing development. The model's accuracy and resilience can be improved by keeping thorough records of the data used, feature engineering methods, and model assessment metrics. In the cybersecurity context, the constant emergence of new malware types necessitates regular updates and testing of the model.(moir, 2023)

TOOLS AND TECHNOLOGY

The design and operation of a JS-MDA online application utilizes a comprehensive range of web tools, machine learning frameworks, and underlying technologies. The integration of numerous tools accelerates the program's creation and improves its performance, accuracy, and scalability.

In developing the frontend of a JS-MDA application, primary technologies are applied. React, a widely-used JavaScript framework, is used to construct dynamic and responsive UI. This is particularly helpful in the context of JavaScript detection, since user interfaces must support complex operations such as file uploads, real-time data scraping, parsing, analyzing, and detection.

Figure 32: Tool and Technology.

The backend of the JS-MDA is managed by server-side logic, data processing, and interaction with machine learning models. Node.js and Express are frequently utilized to design the server architecture. Node.js provides a runtime environment that allows JavaScript to be used for server-side programming. It provides a strong framework for developing RESTful APIs that facilitate communication between the frontend and backend, as well as between the backend and other services.

Moreover, programming environments such as Visual Studio Code (VSCode) and Sublime Text are useful tools in the coding process. They also offer multiple extensions and plugins for code quality assurance and Git integration for version control. Terminal is another tool that gives command-line access to manage dependencies, run scripts, and deploy applications. It provides a direct link to the system, allowing it to perform commands, manage processes, and quickly navigate the file system within the system.

The JS-MDA application system's fundamental functionality rests in its machine-learning capabilities. NumPy is important for numerical computation and data management used for huge multi-dimensional arrays and matrices. Pandas, on the other hand, offers techniques for manipulating data cleaning and preparation. Moreover, Scikit-learn provides simple and efficient tools for data mining and data analysis. Scikit-learn supports a wide array of machine learning algorithms, including classification, regression, and clustering models.

In the machine learning pipeline, data preparation is a key stage. It comprises cleaning the data, tokenizing text inputs, and extracting important attributes. Esprima enables syntactic analysis of JavaScript code. It gives a full Abstract Syntax Tree (AST) representation of the code, which may be utilized for different static analysis purposes. This covers code instrumentation, refactoring, and the creation of code transformation tools. Esprima helps to comprehend and manage the complexity of contemporary online applications. The feature extraction is significant in JavaScript malware detection since it puts raw data into a format acceptable for machine learning algorithms with subsequent modifications.

FINDINGS

- **How can the deployment of a detection engine and a machine learning algorithm assure performance, effictiveness and relaiablity of JS-MDA?**

The development of a detection engine combined with a machine learning algorithm is a JS-MDA web application. The detection engine functions by integrating a trained machine learning model to recognize the type of Javascript code. A trained machine learning model, helps to find malicious javascripts by recognizing patterns from regular benign behavior. This capability plays an important role in handling various sorts of javascript. new malicious scripts could evade normal signature-based detection techniques, but using code analysis prior to implementing machine learning methods works well.

To assure the success of the JS-MDA web application, the machine learning detection model incorporates rigorous training and testing phases. The models are exposed to a wide range of JavaScript content. Techniques like cross-validation are used to analyze the model's performance, allowing for the detection and mitigation of overfitting and underfitting difficulties. The deployment process comprises the fine-tuning of model parameters and the selection of the most suitable methods, These methods directly influence the system's dependability.

The simultaneous integration of the detection engine and machine learning algorithms also addresses the system's processing efficiency and scalability. The infrastructure supporting the application is robust and scalable enough to accommodate changing circumstances. Furthermore, continuous monitoring and updating of the machine learning models can be performed manually to absorb new data and adapt to evolving threats. This update process is important for retaining the system's effectiveness over time.

Reliability is further validated by comprehensive testing and validation procedures. Before deployment, the system undergoes rigorous testing in controlled settings to find and correct any potential vulnerabilities or defects. Evaluations of the overall accuracy and performance of the machine learning models are performed to identify the best model fit for the accurate prediction

Therefore, a JS-MDA web application is a sophisticated applicaiton that includes careful consideration of performance, effectiveness, and reliability. By combining detection performance with machine learning's adaptive capabilities, the system can effectively identify a variety of javascripts. The training, testing, and validation processes assure that the models are accurate and trustworthy, The optimization and scalability concerns ensure that the system can meet real-world needs. Continuous updates and security processes further increase the application's robustness and resilience. The successful deployment of JS-MDA within EWI gives a robust tool for JavaScript detection and analysis, promising long-term support until the JavaScript-based apps are produced and utilized in the real world.

- **What tools and technology can be used to design and develop a JS-MDA web application to integrate with the existing enterprise web infrastructure?**

Designing and developing a Javascript malware detection web application involves a properl selection of tools and technologies. It involves utilizing front-end, back-end technology, and machine learning models, to provide an effective and dependable solution.

React.js is a popular framework for creating web apps. It provides versatility and a broad ecosystem while maintaining user interfaces. It is a component-based architecture, offers reusable components for modularity and maintainability in application design. It is adaptable with other libraries and frameworkst. React-based applications are more business-friendly and ideal for both small and large organizations.

Similarly, Express.js, is a web app framework for designing server-side applications. Node.js is a lightweight and versatile web application framework that is practical for designing server-side applications. Node.js, along with Express.js, serves as the primary framework. Node.js is noted for its non-blocking, event-driven design. It is helpfull for developing scalable network applications. Express.js, a lightweight web framework for Node.js, offers a powerful collection of functionalities. It streamlines the development process by offering HTTP utility methods and

middleware, making it simpler to design a dynamic API that can handle numerous routes and requests from the front-end.

Additionally, Python is an important tool that has comprehensive libraries necessary for machine learning. NumPy supports large, multi-dimensional arrays and matrices, as well as a set of mathematical functions. Pandas are important for data manipulation and analysis. Scikit-learn, a Python machine learning toolkit, is for data mining and data analysis. To display data and outcomes, Matplotlib is applied. This Python package allows for the creation of static, visualizations. These visualizations are needed for analyzing the outputs of the JavaScript malware detection process.

Furthermore, Esprima plays a vital role as a static code analysis tool. It is a high-performance ECMAScript parsing infrastructure, and is used for syntactic analysis of JavaScript code,by parsing code into its syntactic structure.

The integration of the machine learning model into the back-end requires establishing APIs that can receive requests from the front-ends. Flask, a micro-web framework developed in Python, is used for this purpose. Flask may be leveraged to develop RESTful APIs that communicate with the machine learning model, capabile for real-time malware detection. The Python scripts to deliver logic and functions to the front end are supported by Flask.

Development environment tools like Visual Studio Code (VSCode) and Sublime Text are regularly used in Integrated Development Environments (IDEs). These features can boost productivity and code quality. Anaconda is another key tool for managing Python packages and environments. It streamlines the process of package management and deployment, offering a trustworthy environment for data science and machine learning applications.

Furthermore, connecting the JS-MDA web application with the entrprise infrastructure requires compatibility and connection with current systems. This involves integrating with firewalls, intrusion detection systems (IDS), and intrusion prevention systems (IPS) to strengthen the overall security posture. APIs and webhooks may be used to provide real-time communication between the JS-MDA program and these security systems, automating the reaction to discovered threats. However, the scope of the application limits itself to the design and development alone, without real-world integration and deployment. Therefore, the JS-MDA web application is developed with the utilization of proper tools and technologies only for testing and learning purposes.

- **What are the ethical and legal considerations with the implementation of JS-MDA, a web based javascript malware detection applicaiton?**

Implementing a JS-MDA detection includes a difficult ecosystem of ethical and legal considerations. These are crucial for ensuring the application's responsible use, protection of user rights, and conformation to legal standards. Privacy and data protection are two of the key

challenges. It is important to comply with the General Data Protection Regulation GDPR, CCPA and Nepalese law. The acquired data and its application raise ethical concerns.

Additionally, the application must comply with a set of legal standards beyond data protection. Comply with software licensce agreements and maintain intellectual property rights. Third-party libraries and tools must be utilized in compliance with their specific licenses. Failure to comply with these licenses may result in legal action and harm the application's reputation. Moreover, the application should not engage in criminal, unauthorized monitoring of user systems or accessing private information without particular consent.

Accountability is a fundamental component of both ethical and legal compliance. The authors must provide clear governance procedures to control its deployment and use. Setting obligations for data protection authorities, security specialists, and stakeholders. It should include periodic audits, reporting, and the capacity to monitor and react to complaints.

Additionally, the algorithms must be evaluated to ensure conclusion based on unbiased data. It requires adding systems to recognize and minimize bias. . Transparency is critical in AI decision-making procedures Furthermore, the authors of the JS-MDA software must provide clear governance procedures to control its deployment and use. The information about the tool's capabilities must be made transparent. It needs clear, straightforward, and accessible terms of service and privacy policies.

However, from a legal and ethical standpoint, the application's security and reliability is an important factor. Users rely on the application used to identify malware to prevent any losses and damage. The application must undergo rigorous security testing to find and resolve any vulnerabilities. Ethical obligation also entails releasing timely updates and patches to address newly uncovered risks and vulnerabilities. The ethical and regulatory frameworks change continuously, with changes surrounding the JS-MDA application. Continuous ethical analysis and adaptation are needed to address developing issues and incorporate new regulatory duties. It involves staying updated on changes in legislation. The best practice is to interact with stakeholders, and gather feedback from users. To implement the improvements to the privacy rules, security measures, and user interfaces that are important to preserve compliance and trust.

PROJECT AND ISSUE MANAGEMENT

ID	Task	Start Date	Due Date	Duration	2024 J	F	M	A	M	J	J	A	S	O	N	D
1	JS-MDA web application design and development	05/01/2024	08/10/2024	3.5 months												
1.1	Resource Collection	05/02/2024	05/20/2024	18 days												
1.2	Research and Study	05/10/2024	06/10/2024	1 month												
2	Project Proposal	06/10/2024	06/30/2024	20 days												
2.1	Documentation	06/25/2024	06/30/2024	5 days												
3	Planning Phase	05/20/2024	05/30/2024	10 days												
4	Designing Phase	06/01/2024	06/20/2024	20 days												
5	Development Phase	06/10/2024	07/30/2024	1.5 months												
6	Testing Phase	07/15/2024	07/25/2024	10 days												
7	Feedback and Update	07/20/2024	07/25/2024	5 days												
8	Integration Phase	06/30/2024	07/30/2024	1 month												
9	Documentation	07/01/2024	07/25/2024	25 days												

Figure 33: JS-MDA Gantt Chart

CRITICAL ISSUES FACED

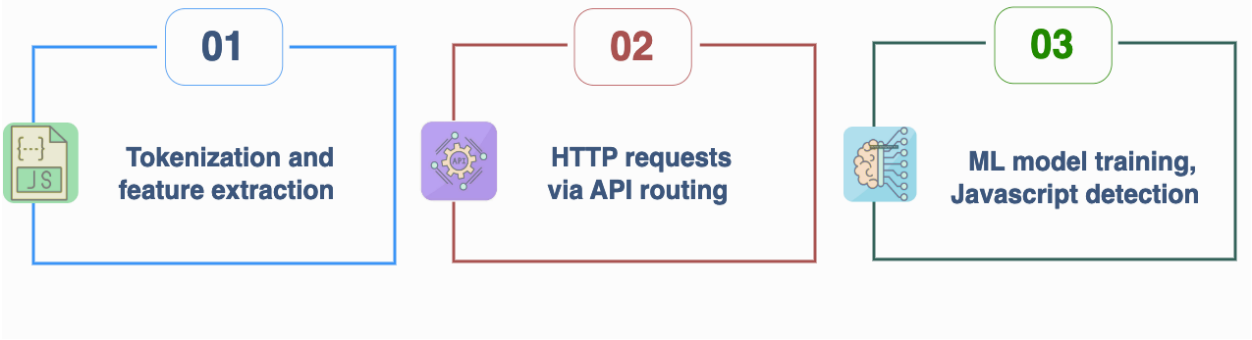


Figure 34: Critical Issues Faced

RISK MANAGEMENT

S.N.	Risks	Occurances	Impact	Axuiliary Plan
1	Availability of technical support	Very high	Disaster	Search for expertise
2	Uncertanity in Technology	high	Disaster	Learn for various sources
3	Application Failure	high	High	Debug application
4	Lack of Predected detection result	high	High	Data optimization
5	ML training	high	Medium	Train 3 to 4 models, evaluate and choose the best
6	Limited Resources	low	Medium	Text books, technical literature, research papers
7	Time Management	low	Medium	Follow schedule
8	Stress & time management	low	Low	Take rest every 2hrs during long hours study

JS-MDA TASK LIST

Task name	Assignee	Due date	Priority	Status
JS-MDA Development workflow				
Thesis title selection	rr rupak rajba...	Apr 1 – 28	High	Complet...
Planning workflow	rr rupak rajba...	Apr 29 – 30	Low	Complet...
Primary resource collection	rr rupak rajba...	May 2 – 20	High	Off track
Research and Study	rr rupak rajba...	May 10 – Jun 10	High	On track
Proposal documentation	rr rupak rajba...	Jul 1 – 6	High	Complet...
Secondary resource collection MERN Stack	rr rupak rajba...	Jun 1	Medium	Complet...
Study and learn design components	rr rupak rajba...	Jun 1 – 7	High	Complet...
Design Frontend components	rr rupak rajba...	Jun 5 – 8	High	Complet...
Test and run the React server	rr rupak rajba...	Jun 7	Medium	Complet...
Study and learn backend components	rr rupak rajba...	Jun 8 – 12	Medium	Complet...
Develop Backend components	rr rupak rajba...	Jun 10 – 13	Medium	Complet...
Test and run Node server	rr rupak rajba...	Jun 14	Medium	Complet...
Develop Middleware components	rr rupak rajba...	Jun 8 – 10	Low	Complet...
Integrate Frontend and Backend	rr rupak rajba...	Jun 14 – 17	Medium	Complet...
Study and learn web scrapping	rr rupak rajba...	Jun 15 – 16	Low	Complet...
develop code and integrate javascript functionality buttons	rr rupak rajba...	Jun 15	Low	Complet...
Study and learn to open, load and save file	rr rupak rajba...	Jun 15	Low	Complet...
develop code and integrate functionality buttons	rr rupak rajba...	Jun 15 – 19	Medium	Complet...
Study and learn javascript parsing, code analysis, static ana	rr rupak rajba...	Jun 17 – 21	Medium	Complet...
develop code and integrate to the functionality buttons for	rr rupak rajba...	Jun 19 – 26	Medium	Complet...
Test and run functionality check	rr rupak rajba...	Jun 25 – 26	Medium	On track
Secondary resource collection Machine Learning	rr rupak rajba...	May 20	Medium	Complet...
Study and Learn Linear Algebra	rr rupak rajba...	May 20 – 31	Medium	Complet...
Study and Learn Machine Learning Algorithms	rr rupak rajba...	May 22 – 31	Medium	Complet...
Study and Learn ML with Python and tools	rr rupak rajba...	May 30 – Jun 1	Medium	Complet...
Collect javascript malicious and benign datasets	rr rupak rajba...	Jun 3 – 5	High	At risk
develop code for data labeling, tokenization, extraction	rr rupak rajba...	Jun 5	Medium	On track
develop code for cleaning, transform, integrate	rr rupak rajba...	Jun 7	Medium	On track
develop code for split, integrate	rr rupak rajba...	Jun 8	Medium	On track
train machine learning model	rr rupak rajba...	Jun 10 – 15	High	On track
test and evaluate machine learning model	rr rupak rajba...	Jun 10 – 15	High	At risk
compare and validate machine learning model	rr rupak rajba...	Jun 10 – 15	High	At risk

Figure 35: Task List

Task name	Assignee	Due date	Priority	Status
Study and learn Flask to integrate ML and python scripts	rr rupak rajba...	Jun 1 – 10	Medium	Complet...
develop scripts with python for button functionalities	rr rupak rajba...	Jun 6 – 19	High	At risk
integrate ML and python scripts to Backend using flask	rr rupak rajba...	Jun 24 – 28	High	At risk
Test and feedback	rr rupak rajba...	Jun 17 – 28	High	On track
check all functionalities	rr rupak rajba...	Jun 19	High	On track
beautify the frontend UI	rr rupak rajba...	Jun 20 – 28	Low	Off track
final thesis documentation	rr rupak rajba...	Jul 1 – Aug 10	Medium	On track
literature review	rr rupak rajba...	Jul 1 – 25	High	Complet...
infographics	rr rupak rajba...	Jul 1 – 15	Medium	Complet...
experiments and results	rr rupak rajba...	Jul 25 – 31	Medium	On track

Figure 35.1: Task List

JS-MDA TIMELINE

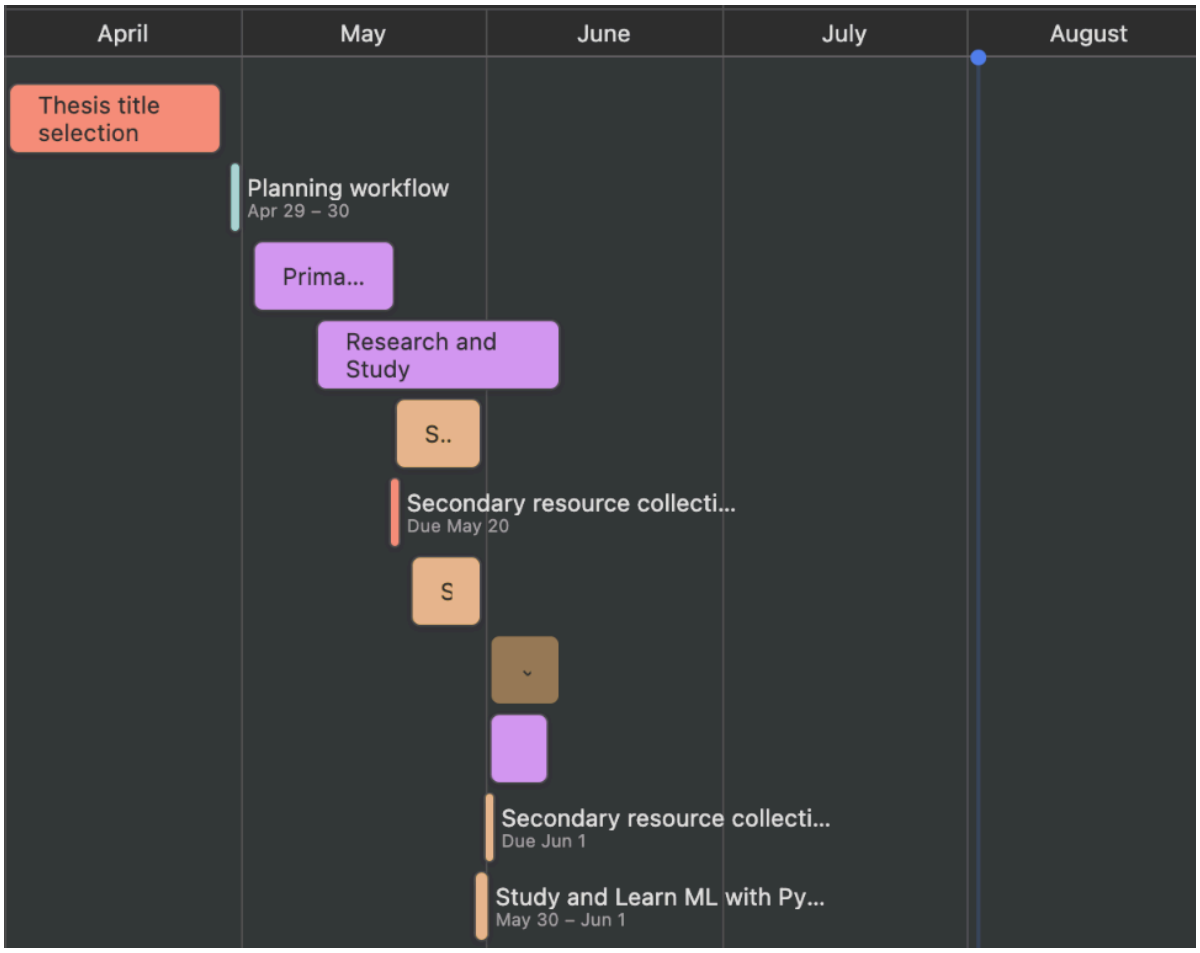


Figure 36: Timeline

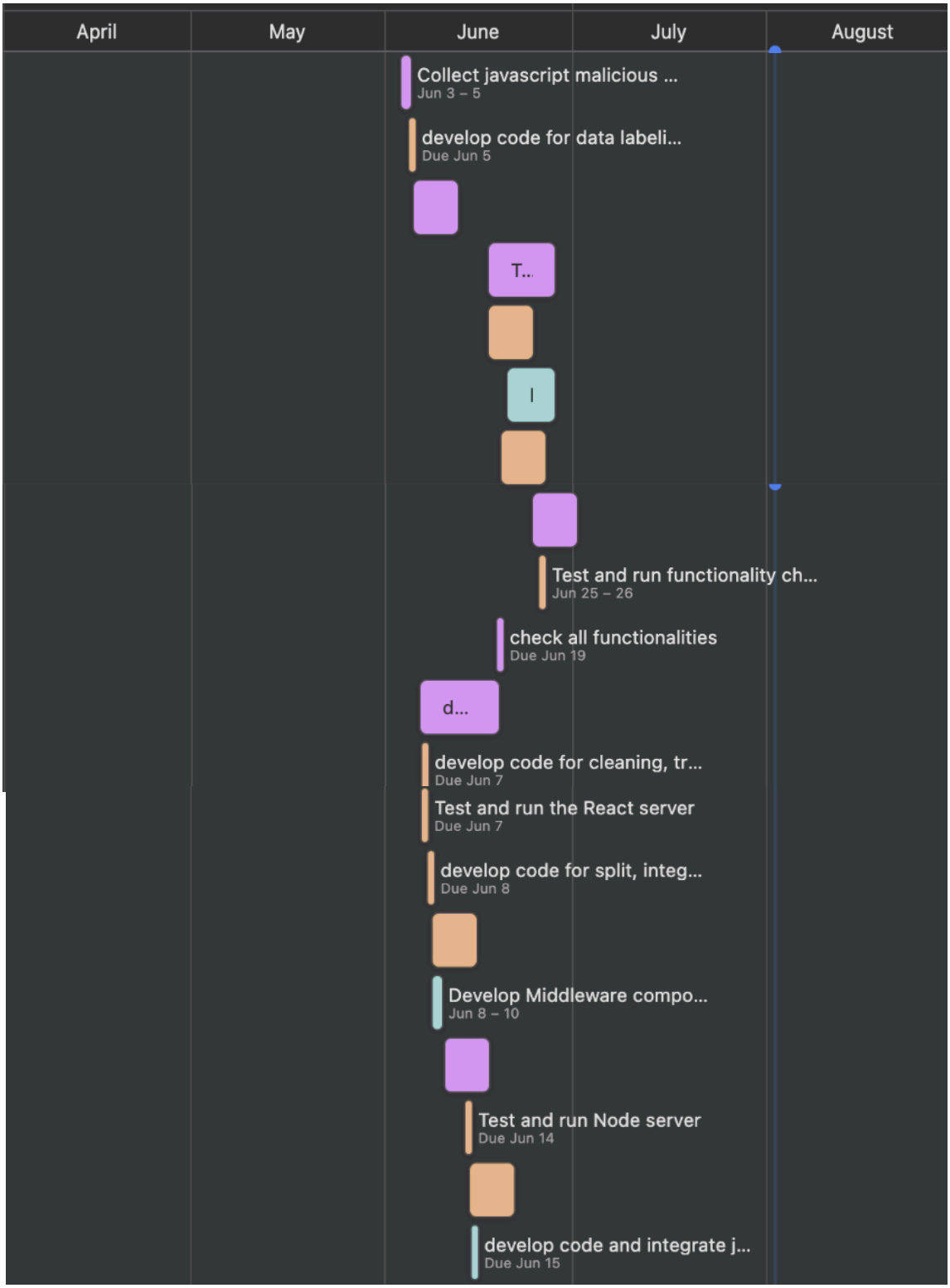


Figure 36.1: Timeline

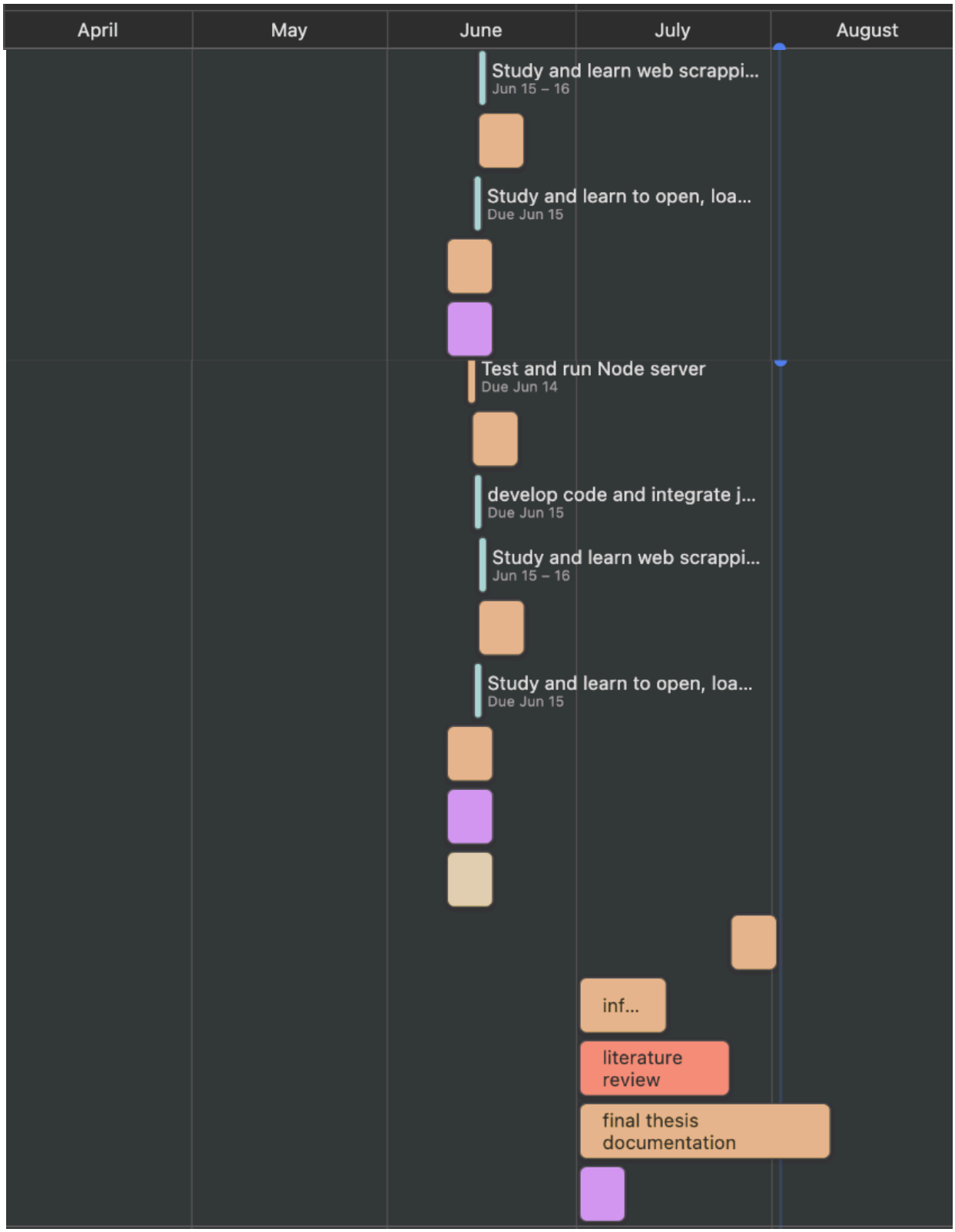


Figure 36.2: Timeline

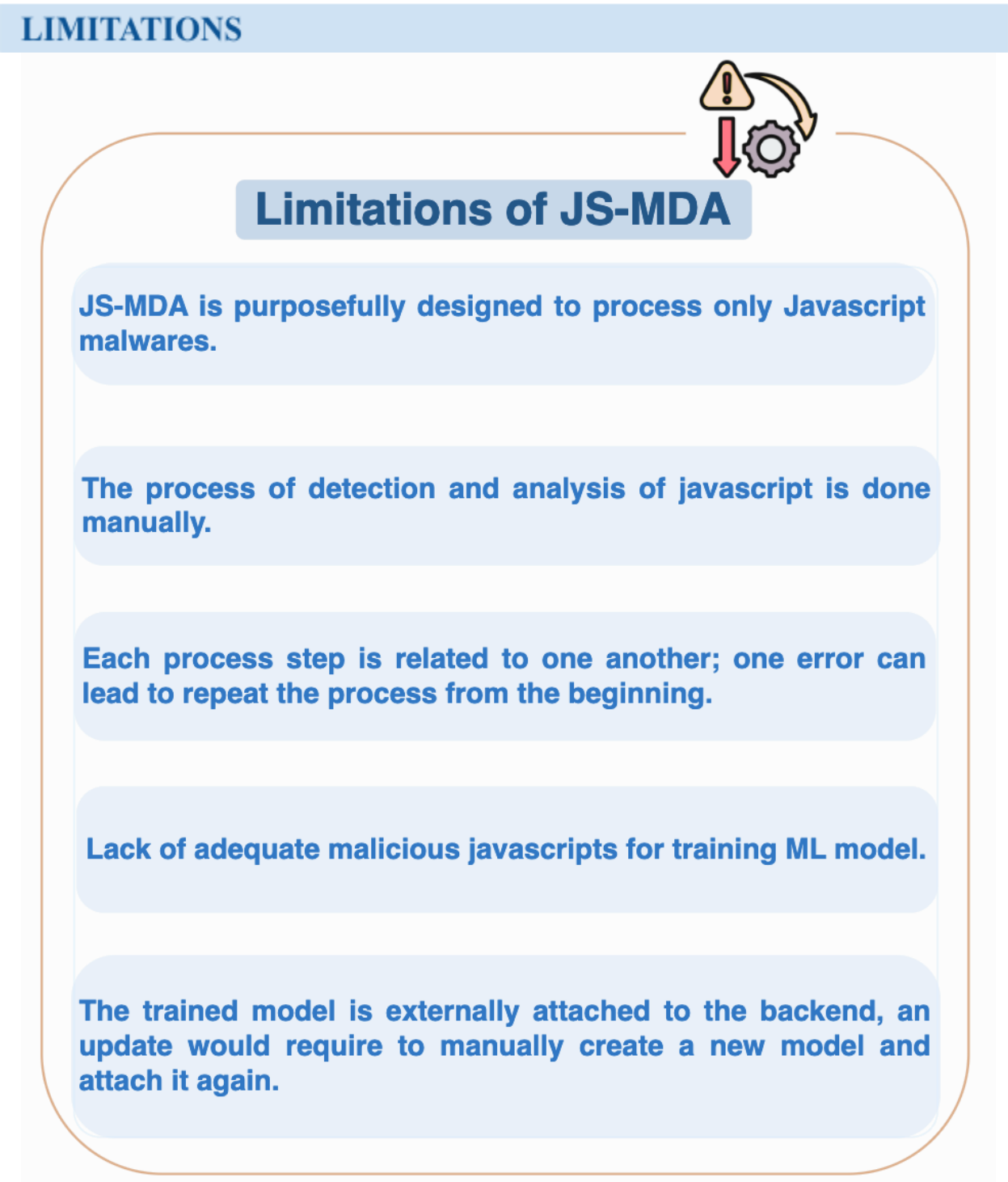


Figure 37: Limitations

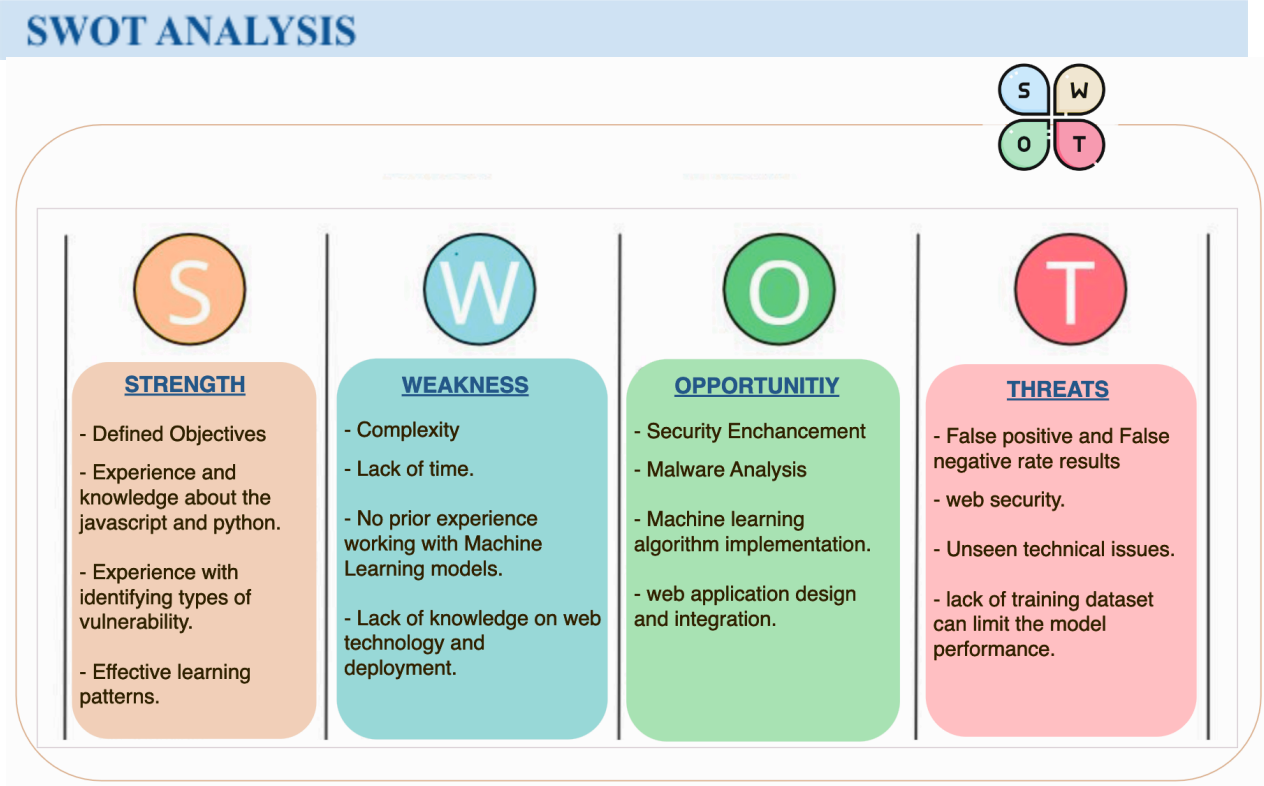


Figure 38: SWOT Analysis

FUTURE WORKS AND RECOMMENDATIONS

The future development of JS-MDA online applications offers significant potential for enhancement particularly with enterprise web infrastructure and the Security Operations Center. User identification, authorization, automated analysis, and advanced machine learning approach are future upgradable options. Each of these areas demands research and study to develop a secure and successful application that fits the requirements of cybersecurity.

Proper user authentication and authorization methods are important to secure the application and its data. So the first work of improvement will focus on combining identity management solutions for enterprises, including single-sign-on (SSO) and multi-factor authentication (MFA). Single Sign-On (SSO) is an authentication technique for users that allows them to quickly access various services with just one set of login credentials. Whereas the MFA provides users with two or more verification elements in order to acquire access. It may involve a mix of the password, 2FA code, and biometric verification.

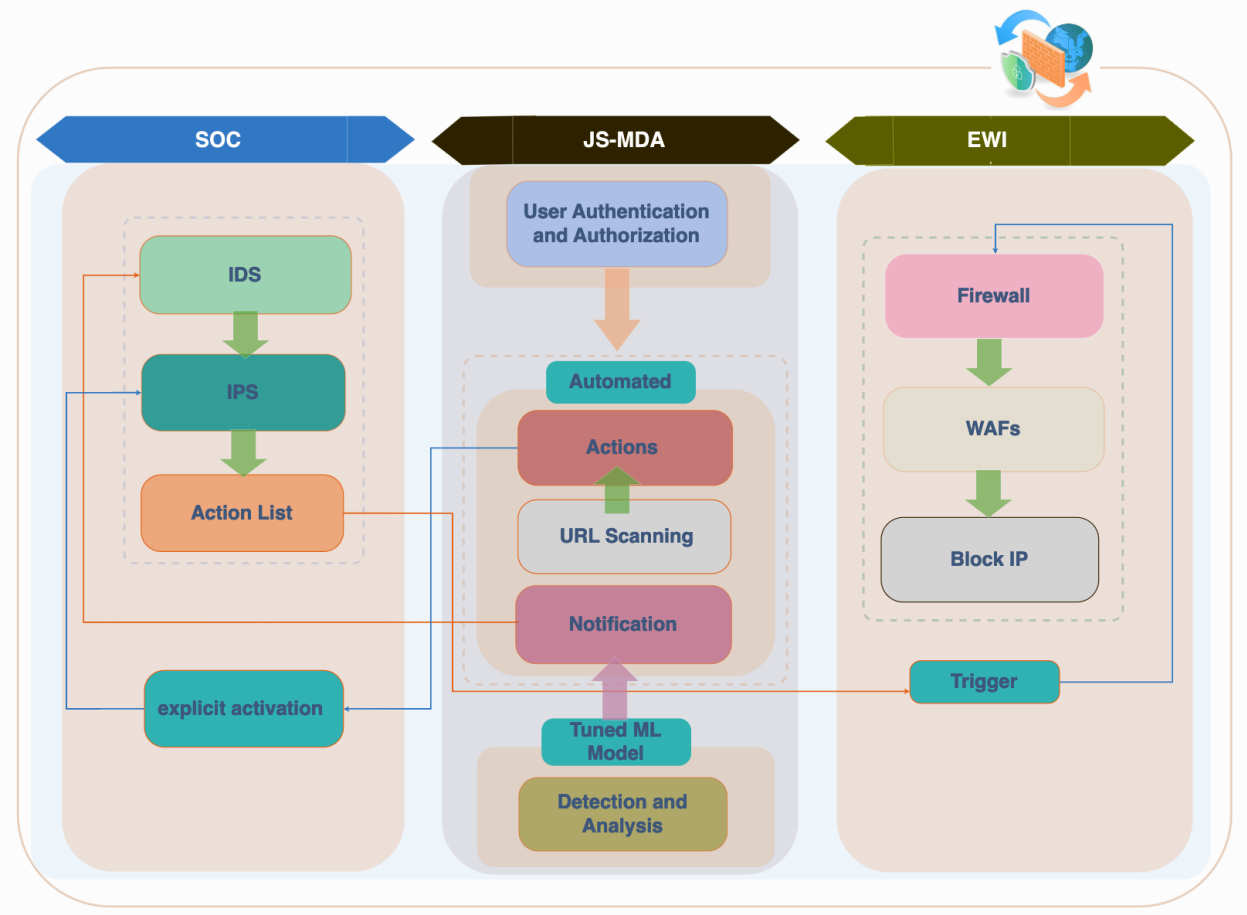


Figure 39: Automated Communications Channels of JS-MDA with EWI and SOC

Another important area for development is the connectivity with the EWI and the Security Operations Center SOC. The objective is to automate the transmission of information and alerts. The program will connect with the firewall systems, IDS, intrusion prevention systems, and web application firewalls (WAFs). It will trigger automated reactions to identified threats, such as blocking IP addresses or triggering actions to isolate critical information systems. Furthermore, notifications will be directed to the SOC. So they can take action, depending on reports. Moreover, the JS-MDA will also be equipped with a suggestion system. This feature will enable SOC analysts to make rapid judgments, therefore boosting the effectiveness of the response process.(Foster City, 2023)

Not only this, automating the analytical procedures of JS-MDA is another key topic. Currently, the approach includes manual stages, including data processing, feature extraction, identification, and analysis. Automating these phases may make the application faster and more accurate. The application may automatically scan incoming URL requests without requiring explicit activatio . for early detection of dangers.

The third part is on boosting the application's capacity to categorize and detect malware based on complicated patterns and anomalies. The model's predictive power will include algorithms that will fine-tune the models based on increasingly sophisticated input from security experts. It will ensure adaptability and overall development of JS-MDA over time.

CONCLUSION

The JS-MDA web application demonstrates the full potential of the MERN stack for machine learning. This JavaScript analysis tool provides ample opportunities for a thorough understanding of the technology and application tools used in its development. The development progressed through substantial study hours and timeless development processes. Furthermore, the application includes functions such as scraping, parsing, code analysis, report generation, and Javascript code identification. All of this contributes to the processing, analysis, and identification of Javascript code types. The JS-MDA architecture handles the application's functions easily. The communication method between the frontend, backend, and flask for Python scripts offers a solid foundation for the application's performance in the browser environment. The inclusion of a detection engine based on machine learning models improves its ability to detect malicious JavaScript code in real time. It also produces reliable results and confirms its efficacy as a JavaScript malware detection tool. Overall, the JS-MDA application demonstrates the use of modern technologies and tools that can work together to develop a highly effective and practical solution for cybersecurity applications that are useful to EWI.

REFERENCES

Edeki, C. (2015) *Agile Software Development methodology*, *European Journal of Mathematics and Computer Science*. Available at: https://www.idpublications.org/wp-content/uploads/2015/05/Agile-Software-Development-Methodology.pdf (Accessed: 16 August 2024).

Foster City, exabeam (2023) *Soc and siem: The role of siem solutions in the SOC*, *Exabeam*. Available at: https://www.exabeam.com/explainers/siem-security/the-soc-secops-and-siem/ (Accessed: 16 August 2024).

moir, sarah (2023) *Documenting machine learning models*, *all posts*. Available at: https://thisisimportant.net/posts/documenting-machine-learning-models/ (Accessed: 16 August 2024).

Javed, M. (2016) *Web application: Operation and maintenance*, *Academia.edu*. Available at: https://www.academia.edu/25862778/Web_application_Operation_and_maintenance (Accessed: 16 August 2024).

AltexSoft, T. (2024) *Technical documentation in software development: Types and T*, *AltexSoft*. Available at: https://www.altexsoft.com/blog/technical-documentation-in-software-development-types-best-practices-and-tools/ (Accessed: 16 August 2024).

Kate Eby, Kate (2019) *All about the iterative design process*, *Smartsheet*. Available at: https://www.smartsheet.com/iterative-process-guide (Accessed: 16 August 2024).

Bhardwaj, A. (2024) *Web application maintenance best practices for 2022*, *Oodles ERP*. Available at: https://erpsolutions.oodles.io/blog/web-application-maintenance-2022/ (Accessed: 16 August 2024).

design, northell (2021) *A Complete Web Development Guide*, *nothell.design*. Available at: https://northell.design/wp-content/uploads/2021/11/A_Complete_Web_Development_Guide_For_Non_Technical_Startup_Founder.pdf (Accessed: 16 August 2024).

Singh, A.K. (2020) *Malicious and benign webpages dataset - data in brief*, *Malicious and Benign Webpage Dataset*. Available at: https://www.data-in-brief.com/article/S2352-3409(20)31198-7/fulltext (Accessed: 16 August 2024).

Kumar, A. (2017) *1: Benign and malware sample statistic | download table*, *A framework for Malware Deteciton with static features using machine learning algorithms*. Available at: https://www.researchgate.net/figure/1-Benign-and-malware-sample-statistic_tbl15_329450675 (Accessed: 16 August 2024).

Grigorev, A. (2021a) *CRISP-DM*, *Machine Learning Bookcamp*. Available at: https://mlbookcamp.com/article/crisp-dm (Accessed: 16 August 2024).

Brown, E. (2014) *Web development with node and Express*, *Web Development with Node & Express*. Available at: https://www.vanmeegern.de/fileadmin/user_upload/PDF/Web_Development_with_Node_Express.pdf (Accessed: 16 August 2024).

Minnick, C. (2022) *Beginning ReactJS Foundations Building User interfaces ...*, *ReactJS Foundations*. Available at: https://dl.ebooksworld.ir/books/Beginning.ReactJS.Foundations.Chris.Minnick.Wiley.9781119685548.EBooksWorld.ir.pdf (Accessed: 16 August 2024).

Corporation, I. (2010) *Application development life cycle: An overview*, *IBM*. Available at: https://www.ibm.com/docs/en/zos-basic-skills?topic=zos-application-development-life-cycle-overview (Accessed: 16 August 2024).

Cyberpedia, C. (2024) *What is a siem solution in a soc?*, *Palo Alto Networks*. Available at: https://www.paloaltonetworks.com/cyberpedia/siem-solutions-in-soc#:~:text=SIEM%20(security%20information%20and%20event,critical%20components%20of%20SOC%20operations. (Accessed: 16 August 2024).

Karaarslan, Tuglular, Sengonca, E., tugkan, Halil (2007) *CITESEERX*, *Enterprise-wide Web Security Infrastructure*. Available at: https://citeseerx.ist.psu.edu/ (Accessed: 16 August 2024).

Hughes, K. (2021) *Going beyond eslint: Overview static analysis in JavaScript*, *Telerik Blogs*. Available at: https://www.telerik.com/blogs/going-beyond-eslint-overview-static-analysis-javascript (Accessed: 16 August 2024).

Smolyakov, V. (2023) *Vsmolyakov/ML_ALGO_IN_DEPTH: ML algorithms in depth*, *GitHub*. Available at: https://github.com/vsmolyakov/ml_algo_in_depth (Accessed: 16 August 2024).

Agarwal, D. (2024) *What is feature extraction and feature extraction techniques*, *Analytics Vidhya*. Available at: https://www.analyticsvidhya.com/blog/2021/04/guide-for-feature-extraction-techniques/ (Accessed: 16 August 2024).

encode, base64 (2010) *Base64 encoding of 'List' - online*, *Base64 Encode*. Available at: https://www.base64encode.org/enc/list/ (Accessed: 16 August 2024).

MozDevNet, M.F. (2024) *FileReader - web apis: MDN*, *MDN Web Docs*. Available at: https://developer.mozilla.org/en-US/docs/Web/API/FileReader (Accessed: 16 August 2024).

Curtsinger, C., Livshits, B., Zorn, B. and Seifert, C. (2011) *Fast and precise in-browser JavaScript malware detection*, *Curtsinger.pdf*. Available at:

https://www.usenix.org/legacy/event/sec11/tech/full_papers/Curtsinger.pdf (Accessed: 16 August 2024).

Erickson, J. (2024) *What is The mern stack?*, *What is the MERN Stack? Guide & Examples | Oracle India*. Available at: https://www.oracle.com/in/database/mern-stack/ (Accessed: 16 August 2024).

Fass, A., Stock, B. and Backers, M. (2019) *JSTAP: A static pre-filter for malicious javascript detection*, *2019 Annual Computer Security* . Available at: https://swag.cispa.saarland/papers/fass2019jstap.pdf (Accessed: 16 August 2024).

V, N. (2024) *What are N-grams and how to implement them in python?*, *Analytics Vidhya*. Available at: https://www.analyticsvidhya.com/blog/2021/09/what-are-n-grams-and-how-to-implement-them-in-python/ (Accessed: 16 August 2024).

Duncan, B. (2022) *Understanding angler exploit kit - part 2: Examining Angler Ek*, *Unit 42*. Available at: https://unit42.paloaltonetworks.com/unit42-understanding-angler-exploit-kit-part-2-examining-angler-ek/ (Accessed: 16 August 2024).

Rieck, K., Krueger, T. and Dewald, A. (2010) *Cujo: Efficient Detection and Prevention of Drive-by- ...*, *2010-acsac.pdf*. Available at: https://mlsec.org/docs/2010-acsac.pdf (Accessed: 16 August 2024).

Hern, A. (2015) *Yahoo users hit by 'Malvertising' campaign*, *The Guardian*. Available at: https://www.theguardian.com/technology/2015/aug/05/yahoo-users-malvertising-campaign-malware (Accessed: 16 August 2024).

Enterprise, E. (ed.) (2012) *Mac trojan infects machines through Microsoft Office exploit - enterprise: Siliconrepublic.com - Ireland's Technology News Service*, *Silicon Republic*. Available at: https://www.siliconrepublic.com/enterprise/mac-trojan-infects-machines-through-microsoft-office-exploit (Accessed: 16 August 2024).

Reed, J. (2024) *How the mac OS X trojan flashback changed cybersecurity*, *Security Intelligence*. Available at: https://securityintelligence.com/articles/how-mac-trojan-flashback-changed-cybersecurity-3/ (Accessed: 16 August 2024).

Perez, T. (2013) *NBC website hacked - be careful surfing*, *Sucuri Blog*. Available at: https://blog.sucuri.net/2013/02/nbc-website-hacked-be-careful-surfing.html (Accessed: 16 August 2024).

Corporation, M. (2024) *CVE-2013-0422*, *NVD*. Available at: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-0422%29 (Accessed: 16 August 2024).

Brewster, T. (2013) *NBC site hacked to serve up malware*, *Silicon UK*. Available at: https://www.silicon.co.uk/workspace/nbc-hacked-malware-exploits-108281 (Accessed: 16 August 2024).

portSwigger, portSwigger (2024) *Server-side vulnerabilities*, *PortSwigger*. Available at: https://portswigger.net/web-security/learning-paths/server-side-vulnerabilities-apprentice (Accessed: 16 August 2024).

Foundation, O. (2022) *Owasp top 10 client-side security risks*, *OWASP Top 10 Client-Side Security Risks | OWASP Foundation*. Available at: https://owasp.org/www-project-top-10-client-side-security-risks/. (Accessed: 16 August 2024).

Maricheva, A. (2023) *History of javascript - read our article to find out!*, *SoftTeco*. Available at: https://softteco.com/blog/history-of-javascript (Accessed: 16 August 2024).

D.Gantz, S. and R.Philpott, D. (2013) *Malware detection*, *Malware Detection - an overview | ScienceDirect Topics*. Available at: https://www.sciencedirect.com/topics/computer-science/malware-detection (Accessed: 16 August 2024).

Solution, H. (2024) *What is infrastructure security?: Glossary*, *What is Infrastructure Security? | Glossary | HPE EUROPE*. Available at: https://www.hpe.com/emea_europe/en/what-is/infrastructure-security.html (Accessed: 16 August 2024).

Academy, portSwigger (2024) *Dom-based vulnerabilities*, *Web Security Academy*. Available at: https://portswigger.net/web-security/dom-based#:~:text=Fundamentally%2C%20DOM%2Dbased%20vulnerabilities%20arise,accessed%20with%20the%20location%20object. (Accessed: 16 August 2024).

Muraya, B. (2023) *Web architecture and Infrastructure Design*, *Medium*. Available at: https://medium.com/@babumuraya/web-architecture-and-infrastructure-design-5fa2e6b7761c (Accessed: 16 August 2024)

Richards, G. *et al.* (2011) *The eval that men do*, *ecoop11.pdf*. Available at: http://janvitek.org/pubs/ecoop11.pdf (Accessed: 16 August 2024).

APPENDIX

Backend Code

server.js

backend > server.js > ...

```
const express = require('express');
const mongoose = require('mongoose');
const cors = require('cors');
const bodyParser = require('body-parser');
const { exec } = require('child_process');

const app = express();

// Middleware
app.use(cors());
app.use(bodyParser.json());

// Connect to MongoDB
/*mongoose.connect('mongodb://localhost:27017/mern-stack-app', { useNewUrlParser: true, useUnifiedTopology: true })
    .then(() => console.log('MongoDB connected...'))
    .catch(err => console.log(err));*/

// API Route for text input (previous example)
app.post('/api/submit', async (req, res) => {
    const { text } = req.body;
    const newText = new Text({ content: text });
    await newText.save();
    res.json({ message: 'Text received!', data: newText });
});

// API Route for scraping
app.post('/api/scrape', (req, res) => {
    const url = req.body.url;
    if (!url) {
        return res.status(400).json({ error: 'URL is required' });
    }

    // Specify the full path to the Python executable
    const pythonPath = '/opt/homebrew/bin/python3.10'; // Update this to your Python path
    const command = `${pythonPath} scrape.py`;


    exec(command, (err, stdout, stderr) => {
        if (err) {
            console.error(`exec error: ${err}`);
            return res.status(500).json({ error: err.message });
        }
        try {
            const output = JSON.parse(stdout);
            res.json(output);
        } catch (parseErr) {
            res.status(500).json({ error: 'Failed to parse Python script output' });
        }
    });
});

// Start the server
const PORT = process.env.PORT || 1000;
app.listen(PORT, () => console.log(`Server started on port ${PORT}`));
```

Figure 40: Server.js

routes.js

backend > routes > routes.js > ...

```
1   const express = require('express');
2   const router = express.Router();
3
4   // Example route
5   router.get('/', (req, res) => {
6     res.send('Get all routes');
7   });
8
9   module.exports = router;
10
```

Figure 41: Routes.js

.env

backend > .env

```
1   FLASK_RUN_PORT=2000
```

Figure 42: .env

package.json

backend > {} package.json > ...

```
1   {
2     "name": "backend",
3     "version": "1.0.0",
4     "main": "index.js",
      Debug
5     "scripts": {
6       "start": "node server.js",
7       "dev": "nodemon server.js",
8       "test": "echo \"Error: no test specified\" && exit 1"
9     },
10    "keywords": [],
11    "author": "",
12    "license": "ISC",
13    "description": "",
14    "dependencies": {
15      "bcrypt": "^5.1.1",
16      "body-parser": "^1.20.2",
17      "cors": "^2.8.5",
18      "dotenv": "^16.4.5",
19      "esprima": "^4.0.1",
20      "express": "^4.19.2",
21      "jsonwebtoken": "^9.0.2",
22      "mongoose": "^8.4.4"
23    },
24    "devDependencies": {
25      "nodemon": "^3.1.4"
26    }
27  }
28
```

Figure 43: Backend package.json

Flask.py 6

backend > Flask.py > upload_file

```python
from flask import Flask, request, jsonify, json
from flask_cors import CORS
from helium import start_chrome
from bs4 import BeautifulSoup
import subprocess
import joblib
import pandas as pd
import os

app = Flask(__name__)
tmp_file_path = "tmp.txt" #Define temporary file path
lexical_file_path = "lexical.csv"
CORS(app)

# Load trained model and LabelEncoders
#pipeline_rf = joblib.load('random_forest_model.pkl')
#le_token = joblib.load('label_encoder_token.pkl')
#le_feature = joblib.load('label_encoder_feature.pkl')
@app.route('/api/saveToTmp', methods=['POST'])
def save_to_tmp():
    try:
        content = request.json['content']
        tmp_file_path = "tmp.txt"

        # Clear the contents of tmp.txt and append the new content
        with open(tmp_file_path, 'w') as tmp_file:
            tmp_file.write(content)

        return {'message': 'Content saved to tmp.txt.'}, 200
    except Exception as e:
        return {'error': str(e)}, 500

@app.route('/api/upload', methods=['POST'])
def upload_file():
    content = request.form.get('content')

    if not content:
        return jsonify({'error': 'Content is required'}), 400

    try:
        # Clear the contents of tmp.txt and append the new content
        with open(tmp_file_path, 'w') as tmp_file:
            tmp_file.write(content)

        return jsonify({'message': 'File content saved to tmp.txt successfully.'}), 200
    except Exception as e:
        return jsonify({'error': str(e)}), 500

@app.route('/api/scrape', methods=['POST', 'OPTIONS'])
def scrape():
    if request.method == 'OPTIONS':
        return jsonify({'allow': 'POST'}), 200

    data = request.get_json()
    url = data.get('url')
    if not url:
        return jsonify({'error': 'URL is required'}), 400

    try:
        browser = start_chrome(url, headless=True)
        soup = BeautifulSoup(browser.page_source, 'html.parser')
        scripts = soup.find_all('script')

```

Figure 44: Flask.py

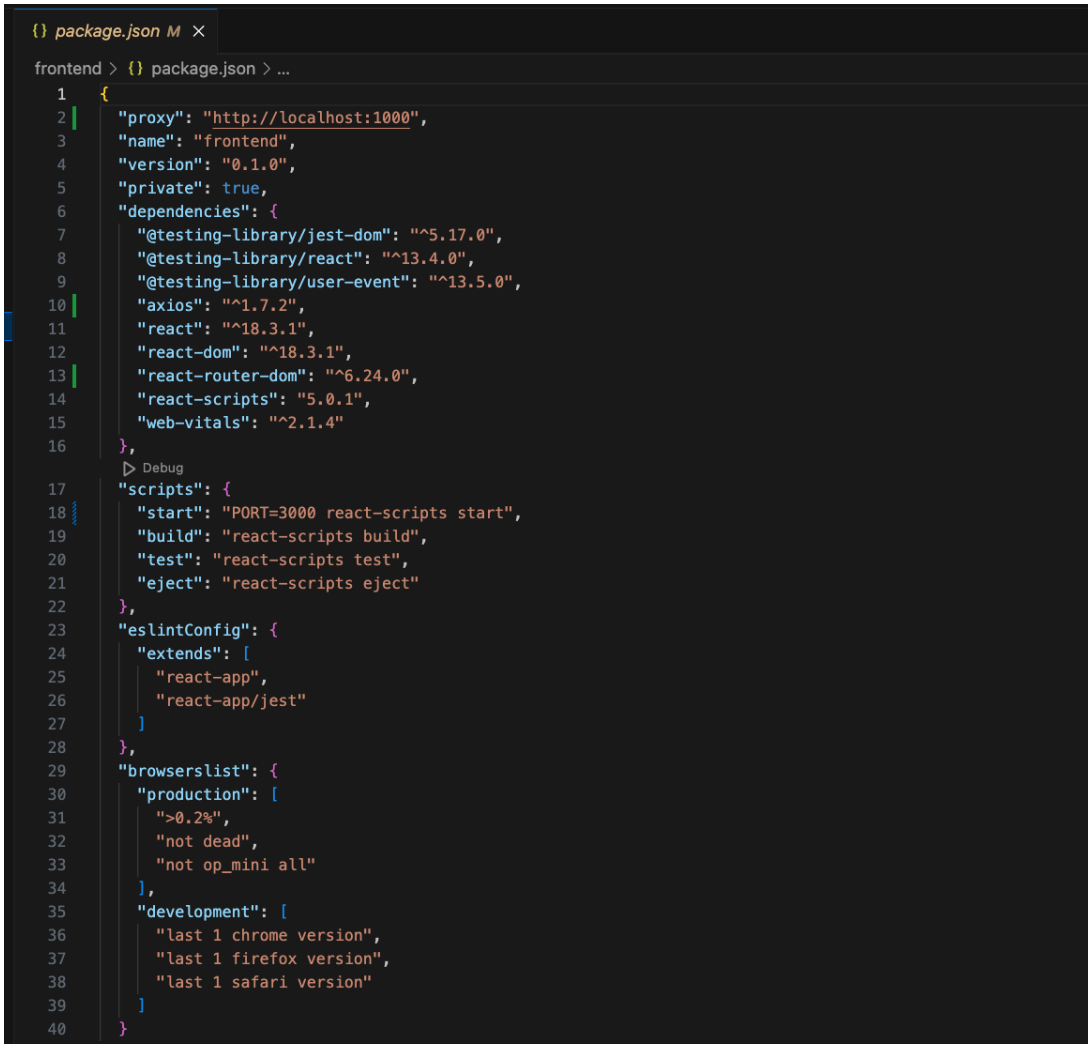
Flask.py 6

backend > Flask.py > generate_report

```python
def identify():
            raise RuntimeError(f"Detect script failed: {result.stderr.strip()}")

        detection_output = result.stdout.strip()

        return jsonify({'detection_output': detection_output})

    except FileNotFoundError:
        return jsonify({'error': 'tmp.txt not found'}), 404
    except Exception as e:
        return jsonify({'error': str(e)}), 500

@app.route('/api/report', methods=['POST'])
def generate_report():
    try:
        data = request.get_json()
        file_path = data.get('filePath', 'tmp.txt')

        result = subprocess.run(
            ["python3.10", "static_report.py", file_path],
            capture_output=True,
            text=True
        )

        if result.returncode != 0:
            raise RuntimeError(f"Report script failed: {result.stderr.strip()}")

        report = result.stdout.strip()
        return jsonify({'report': report})

    except Exception as e:
        return jsonify({'error': str(e)}), 500

@app.route('/api/detect', methods=['POST'])
def detectionengine():
    try:
        data = request.get_json()
        file_path = data.get('filePath', 'lexical.csv')

        # Ensure the file path exists
        if not os.path.exists(file_path):
            raise FileNotFoundError(f"File not found: {file_path}")

        # Run the prediction script with the given file path
        result = subprocess.run(
            ["python", "predection.py", file_path],
            capture_output=True,
            text=True
        )

        if result.returncode != 0:
            raise RuntimeError(f"Prediction script failed: {result.stderr.strip()}")

        report = result.stdout.strip()
        return jsonify({'report': report})

    except Exception as e:
        return jsonify({'error': str(e)}), 500



if __name__ == '__main__':
    app.run(port=2000, debug=True)
```

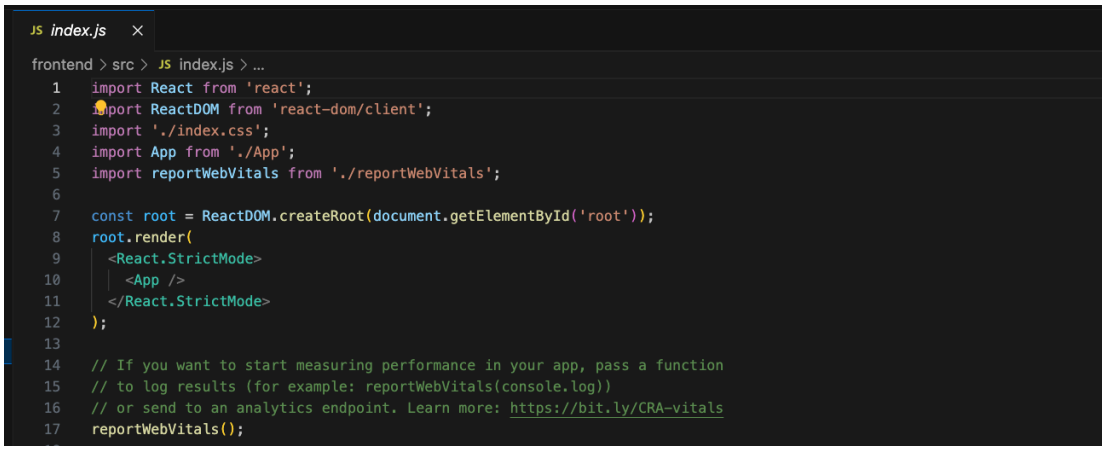
Figure 45.1: Flask.py

Frontend Code



```
{} package.json M ×
frontend > {} package.json > ...
{
  "proxy": "http://localhost:1000",
  "name": "frontend",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.17.0",
    "@testing-library/react": "^13.4.0",
    "@testing-library/user-event": "^13.5.0",
    "axios": "^1.7.2",
    "react": "^18.3.1",
    "react-dom": "^18.3.1",
    "react-router-dom": "^6.24.0",
    "react-scripts": "5.0.1",
    "web-vitals": "^2.1.4"
  },
  ▷ Debug
  "scripts": {
    "start": "PORT=3000 react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
```

Figure 46: Frontend package.json



```
JS index.js ×
frontend > src > JS index.js > ...
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Figure 47: Index .js

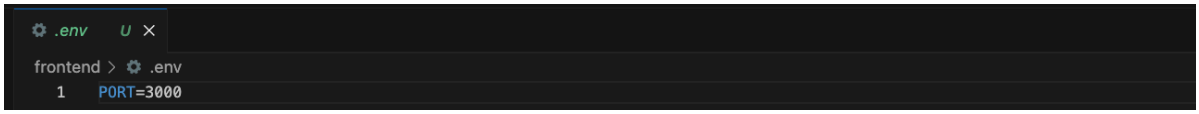


Figure 48: backend .env

Figure 49: App.js

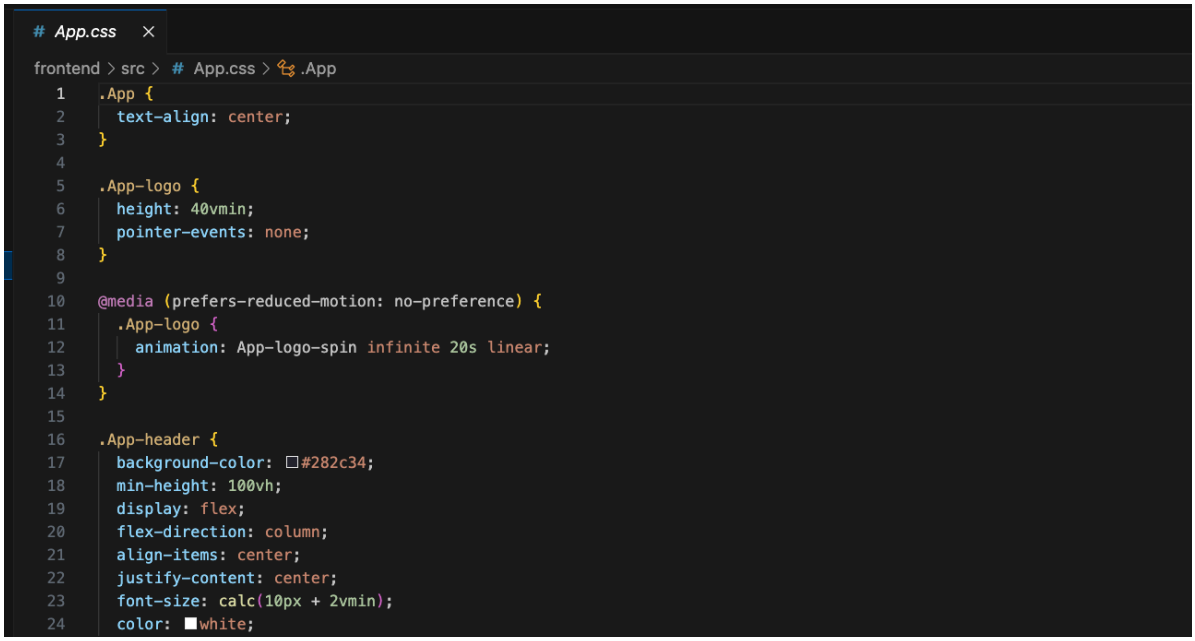


Figure 50: App.css

index.css M

frontend > src > # index.css > input[type="text"]

```css
body {
  margin: 0;
  font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', 'Roboto', 'Oxygen',
    'Ubuntu', 'Cantarell', 'Fira Sans', 'Droid Sans', 'Helvetica Neue',
    sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  background-color: #f8f9fa; /* Light orange background color */
}

.container {
  max-width: 800px;
  margin: 0 auto;
  padding: 20px;
}

.row {
  margin-bottom: 10px;
}

/* Style for form and input */
form {
  background-color: #fff;
  box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
  display: flex;
  align-items: center;
  padding: 15px 20px;
  border-radius: 4px;
}

input[type="text"] {
  flex: 1;
  padding: 8px;
  font-size: 14px;
  border: 1px solid #ced4da;
  border-radius: 4px;
  margin-right: 10px;
}

button {
  padding: 8px 16px;
  font-size: 14px;
  background-color: #007bff;
  color: white;
  border: none;
  cursor: pointer;
  border-radius: 4px;
  margin-left: 10px;
}

button:hover {
  background-color: #0056b3;
}

/* Style for file upload */
label {
  display: block;
  margin-top: 10px;
  font-size: 14px;
}
```

Figure 51: index.css.

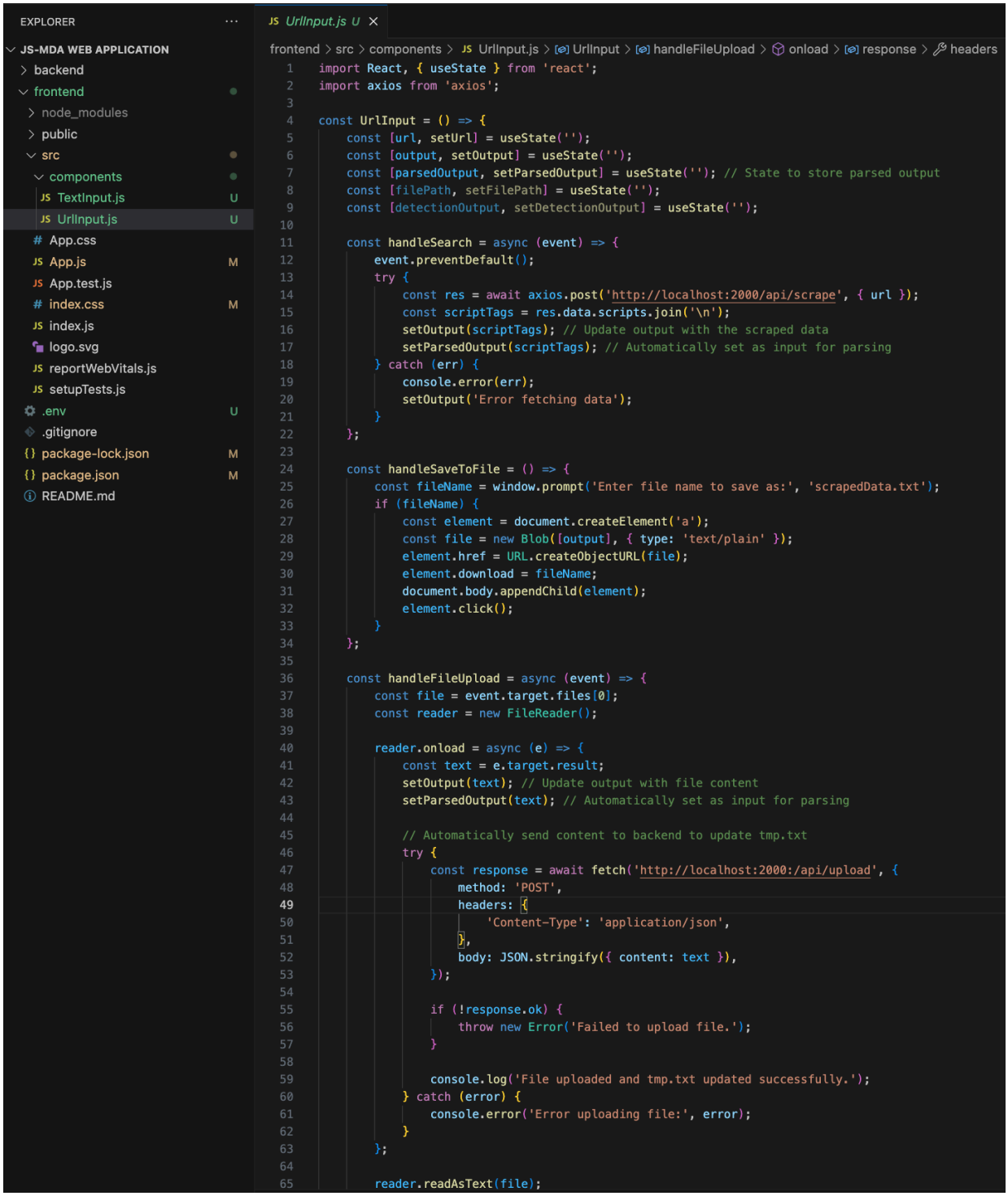


Figure 52: UrlInput.js

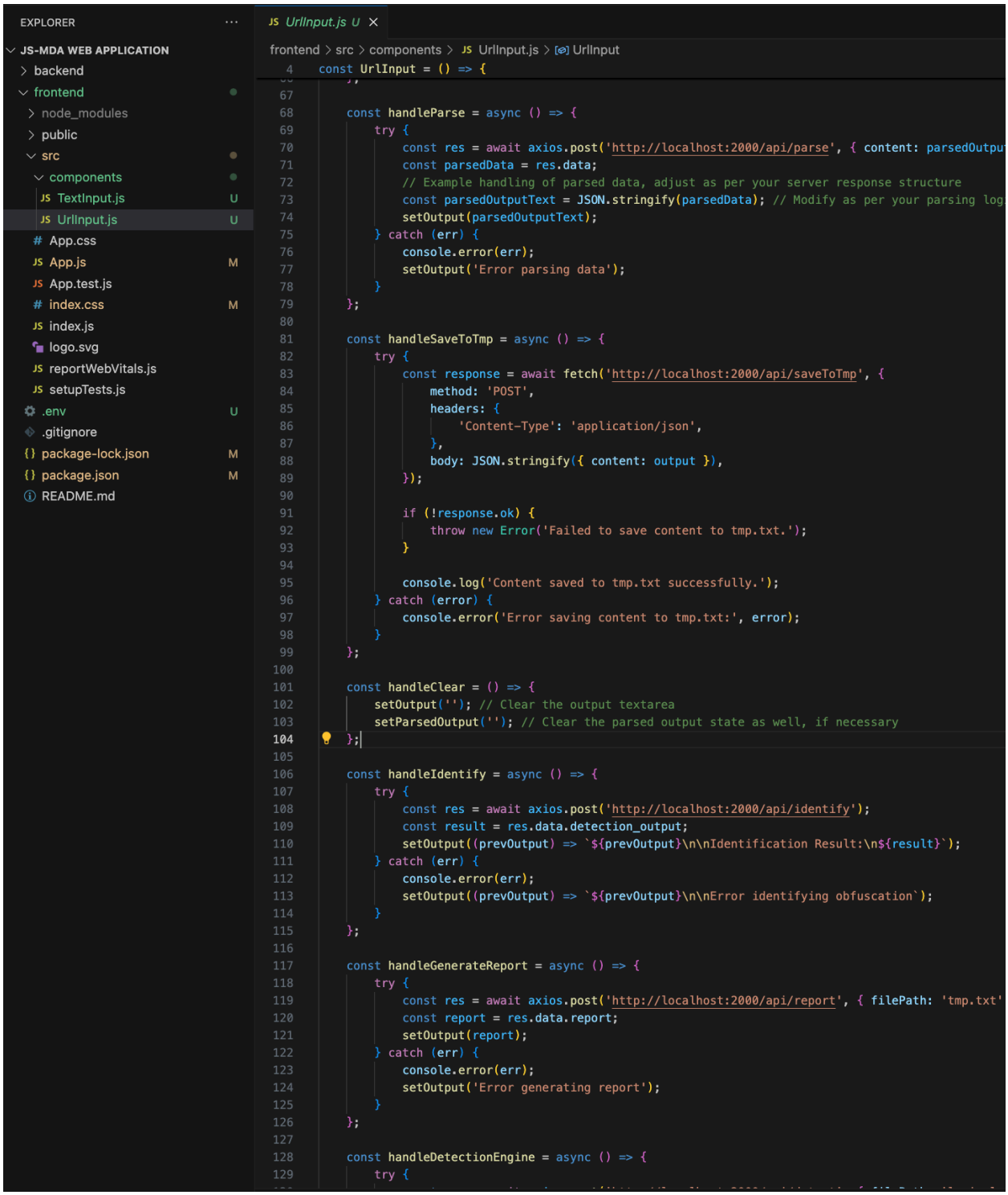


Figure 51.2:UrlInput.js

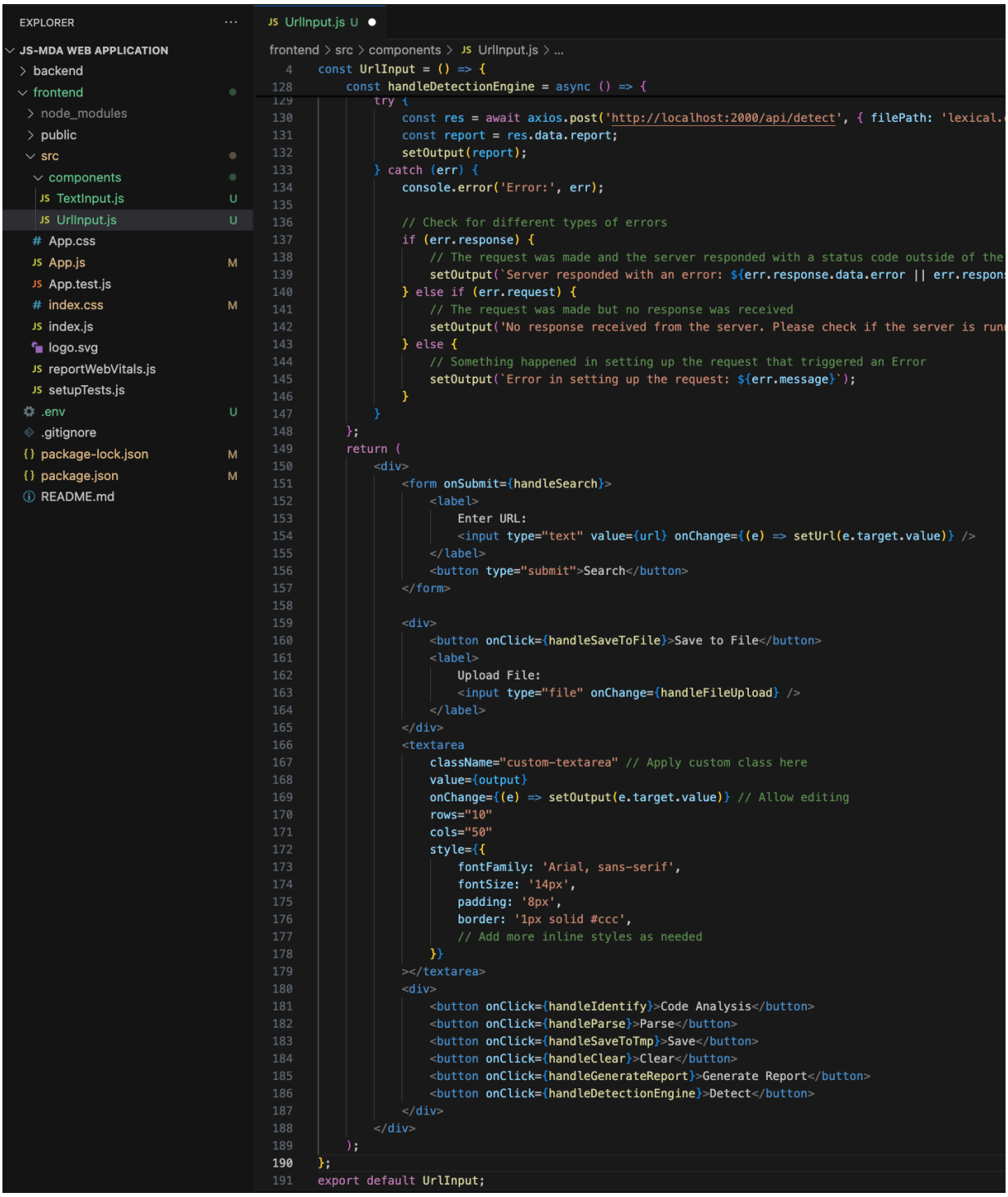


Figure 51.5: UrlInput.js

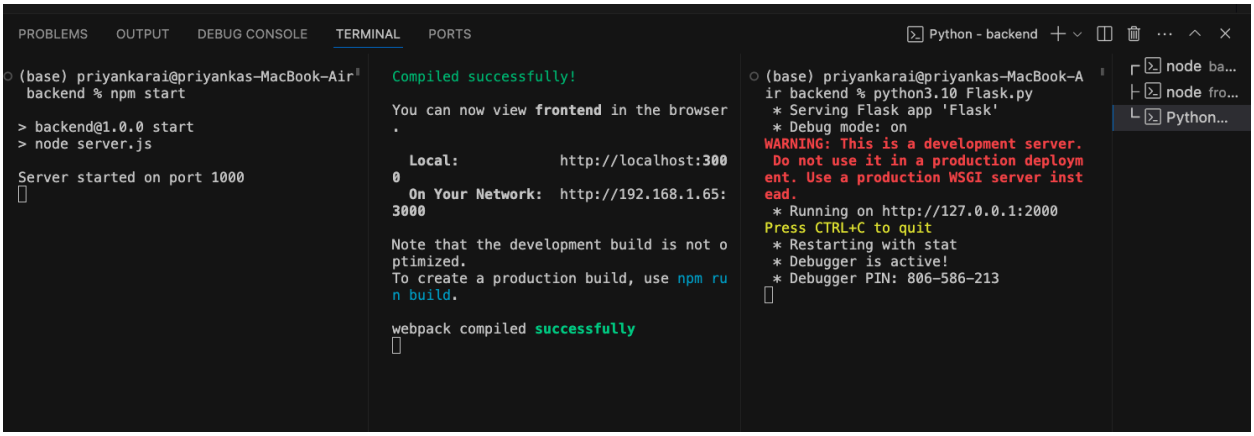


Figure 52: Node server, React server, Flask server

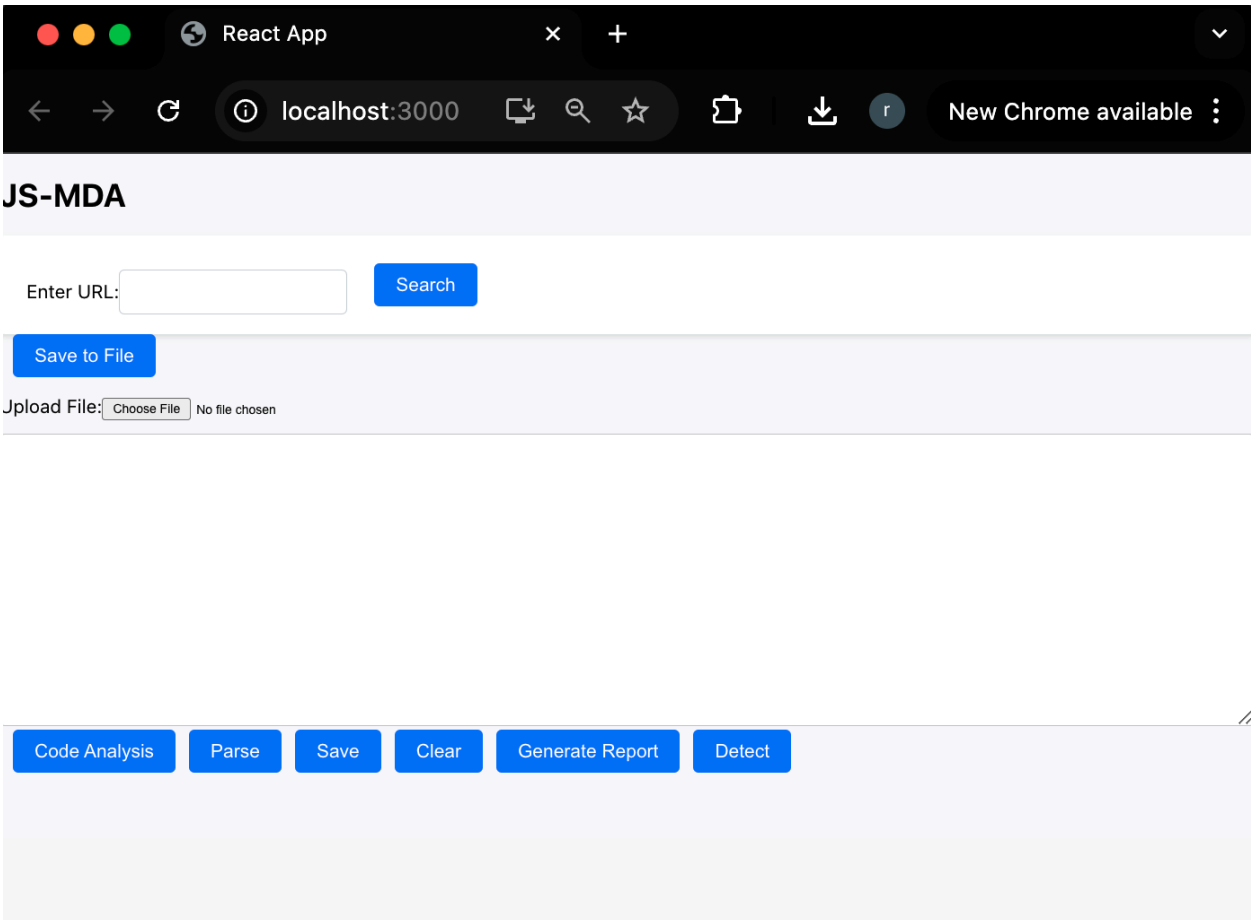


Figure 53: JS-MDA User Interface

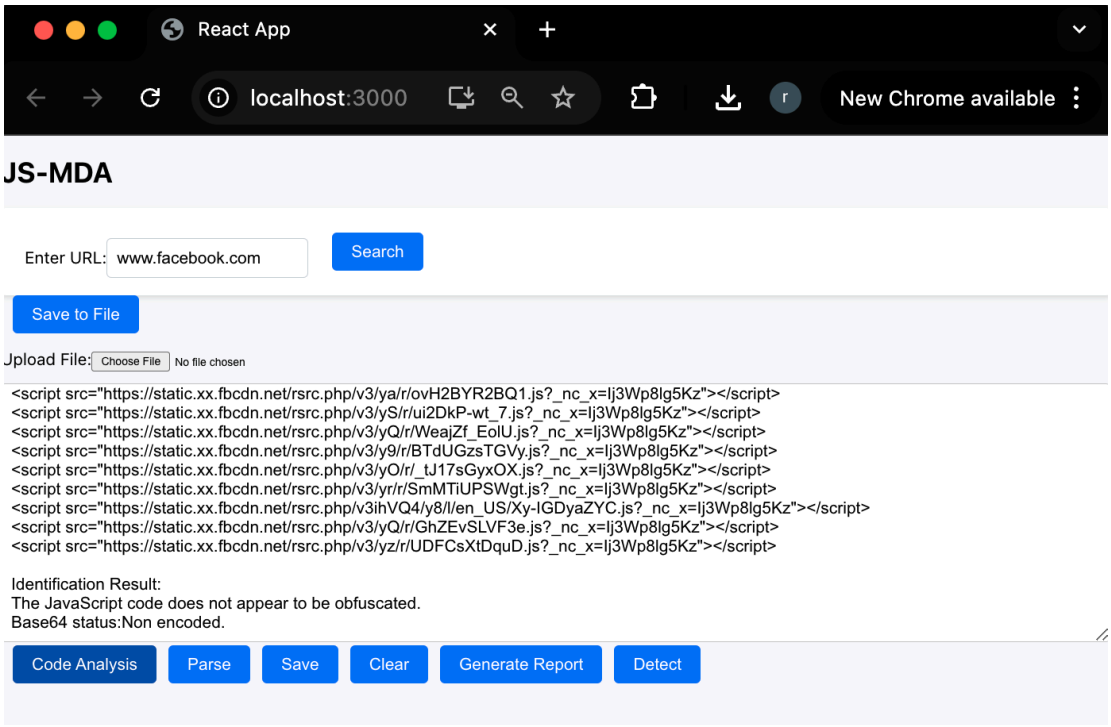


Figure 54: Javascript Scraping

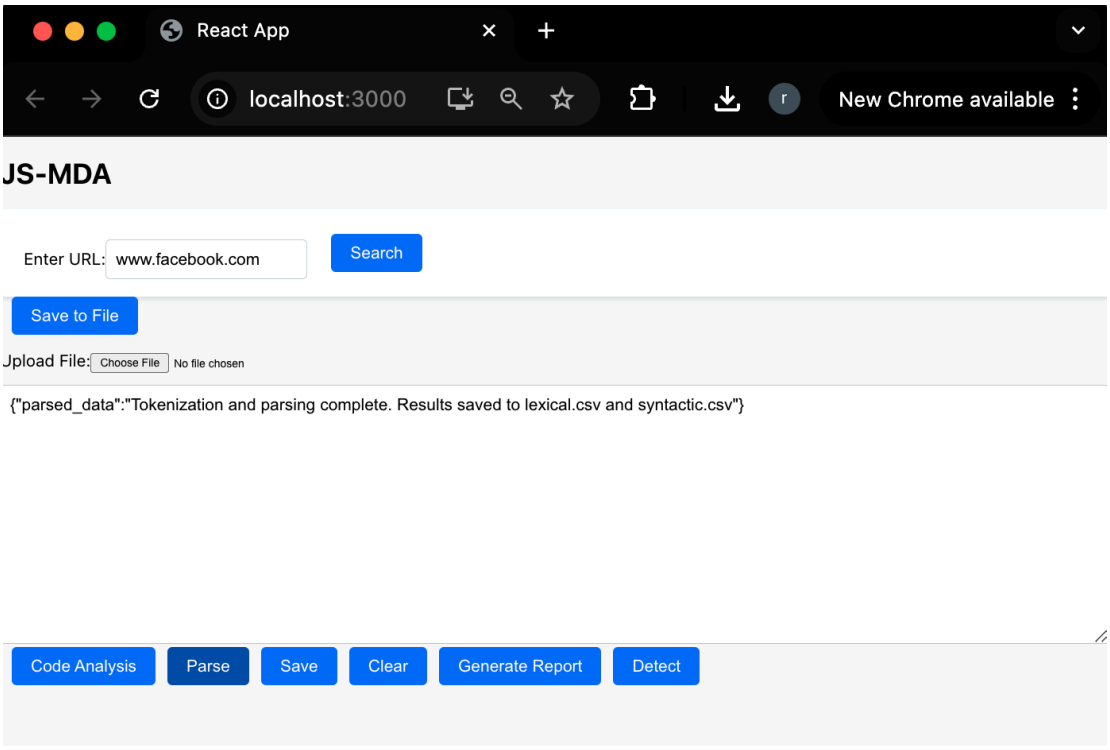


Figure 55: Parsing the script

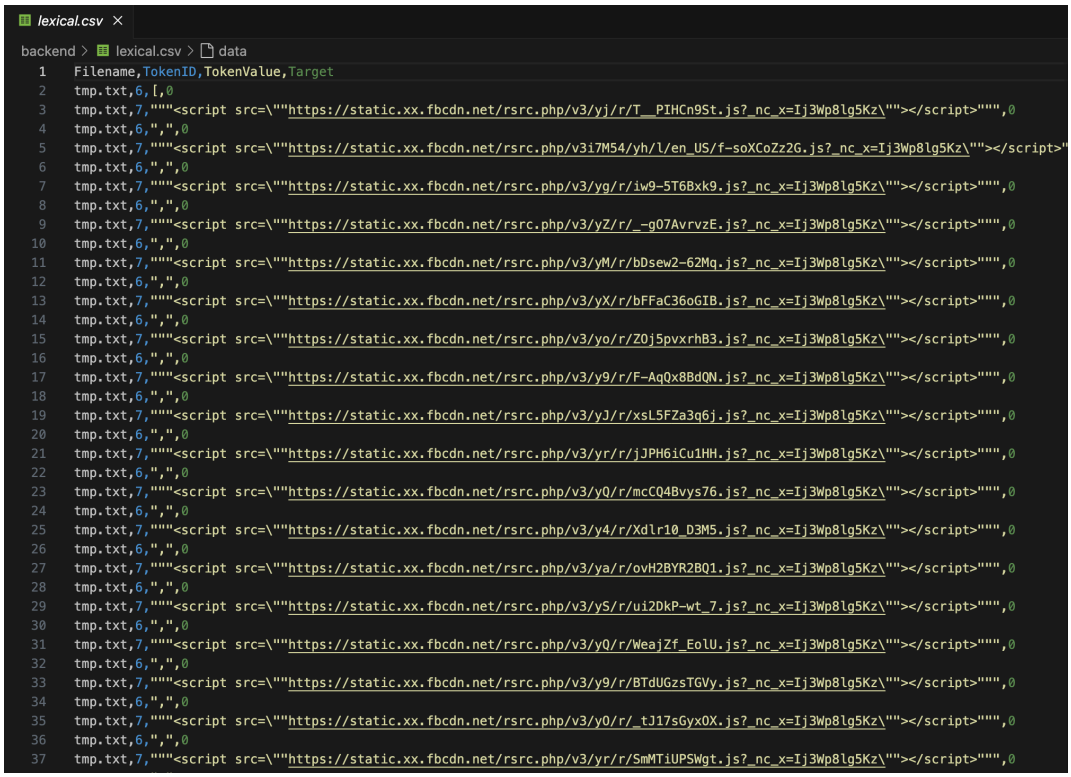


Figure 56: Tokenized data

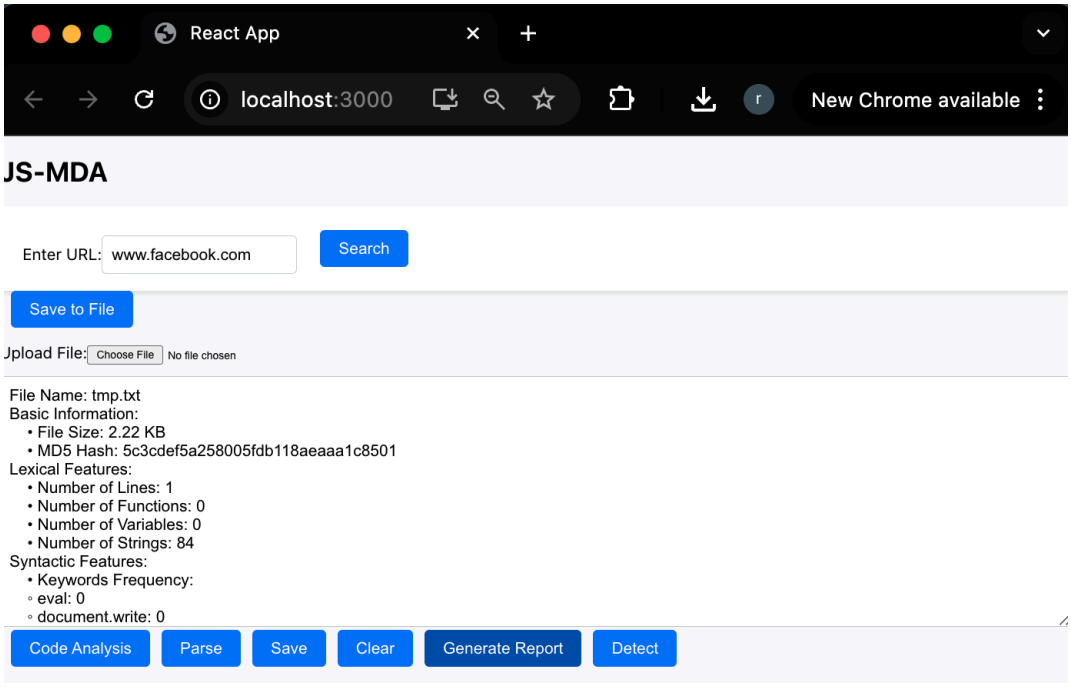


Figure 57: Static Analysis Report

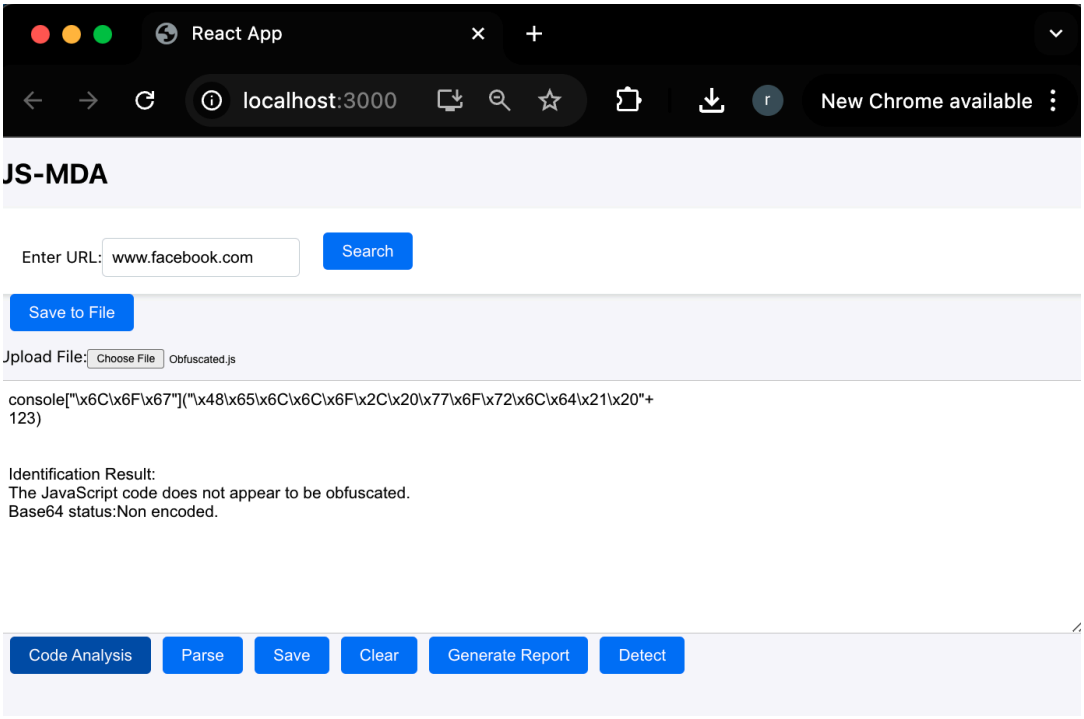


Figure 68: Code Analysis for Obfuscation

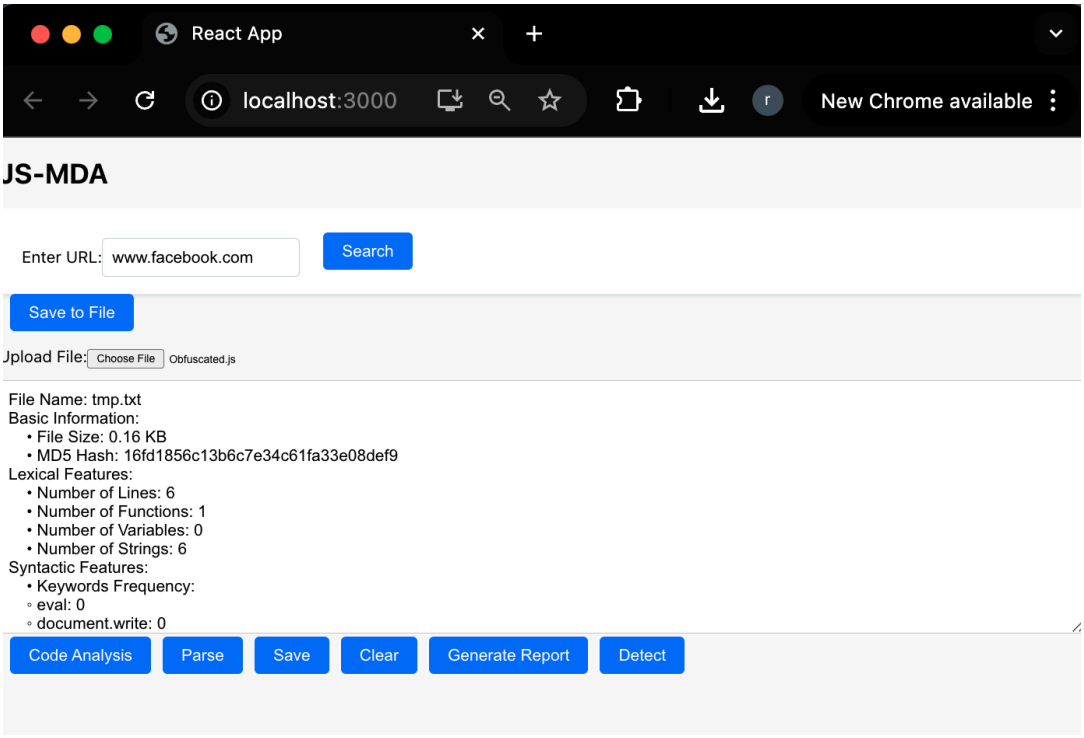


Figure 69: Obfuscated.js file static report

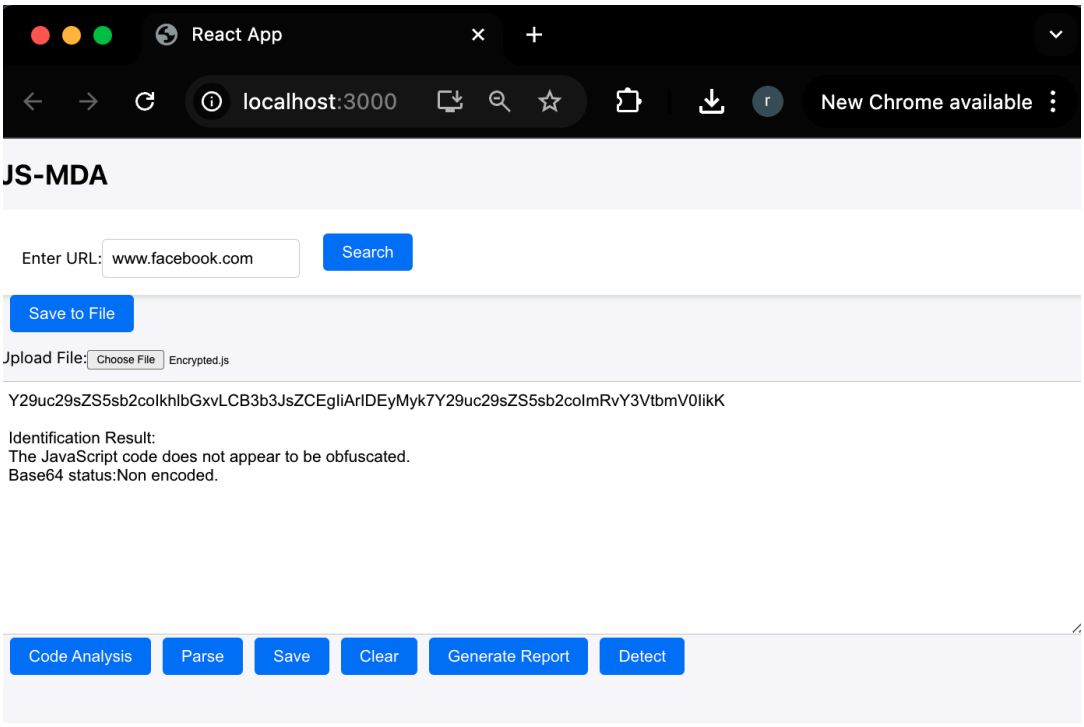


Figure 70: Code Analysis for Encryption

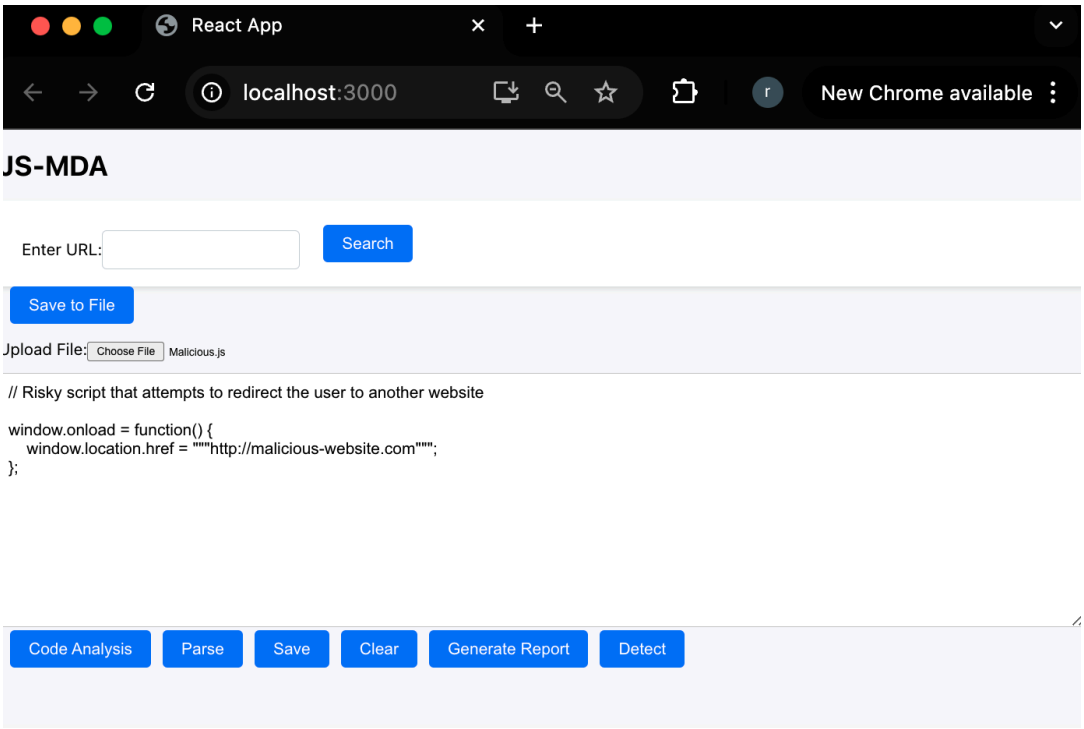


Figure 71: Loading Malicious.js file

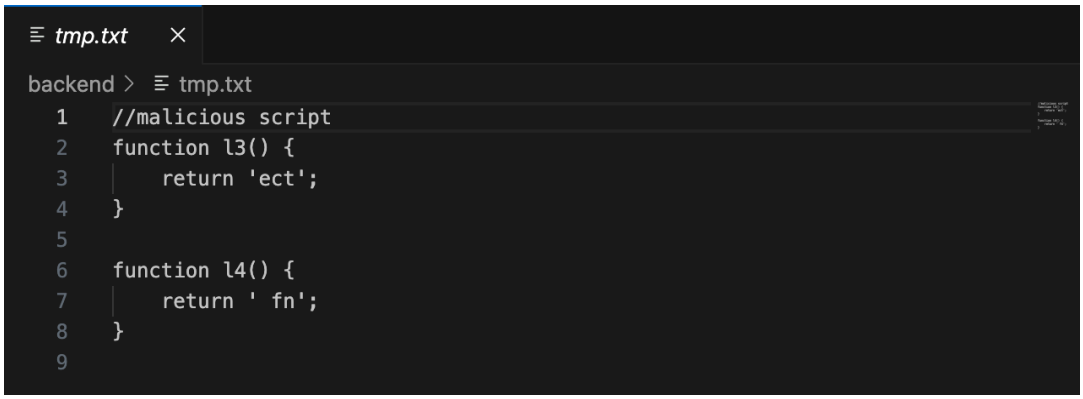


Figure 72: Content of file loaded to tmp.txt in the backend

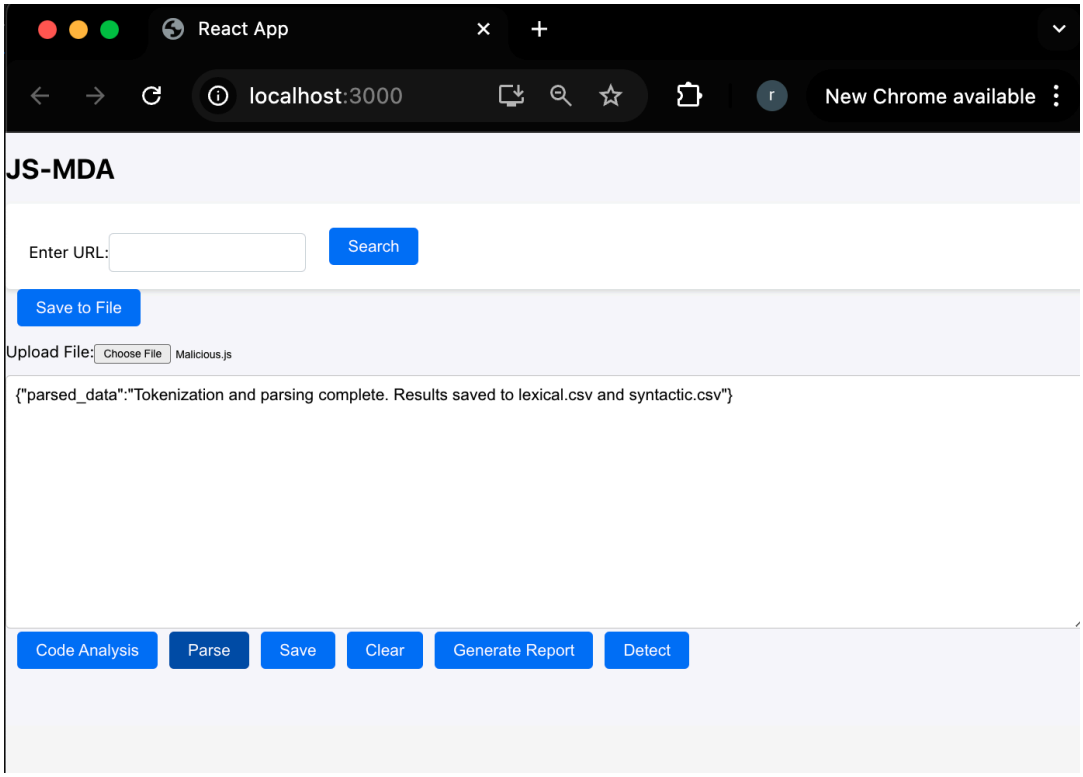


Figure 73: Parse the data

backend > lexical.csv > data

```
Filename,TokenID,TokenValue,Target
tmp.txt,3,function,0
tmp.txt,2,l3,0
tmp.txt,6,(,0
tmp.txt,6,),0
tmp.txt,6,{,0
tmp.txt,3,return,0
tmp.txt,7,'ect',0
tmp.txt,6,;,0
tmp.txt,6,},0
tmp.txt,3,function,0
tmp.txt,2,l4,0
tmp.txt,6,(,0
tmp.txt,6,),0
tmp.txt,6,{,0
tmp.txt,3,return,0
tmp.txt,7,' fn',0
tmp.txt,6,;,0
tmp.txt,6,},0
tmp.txt,3,function,0
tmp.txt,2,l3,0
tmp.txt,6,(,0
tmp.txt,6,),0
tmp.txt,6,{,0
tmp.txt,3,return,0
tmp.txt,7,'ect',0
tmp.txt,6,;,0
tmp.txt,6,},0
tmp.txt,3,function,0
tmp.txt,2,l4,0
tmp.txt,6,(,0
tmp.txt,6,),0
```

Figure 74: Tokenized data

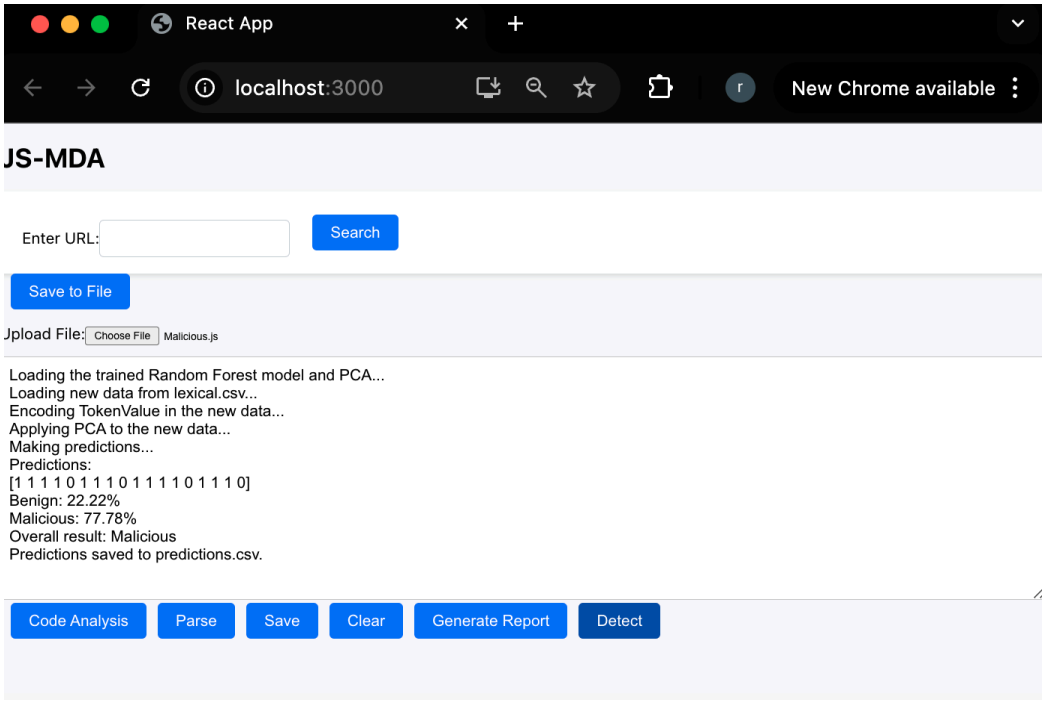


Figure 75: Classifier prediction as Malicious

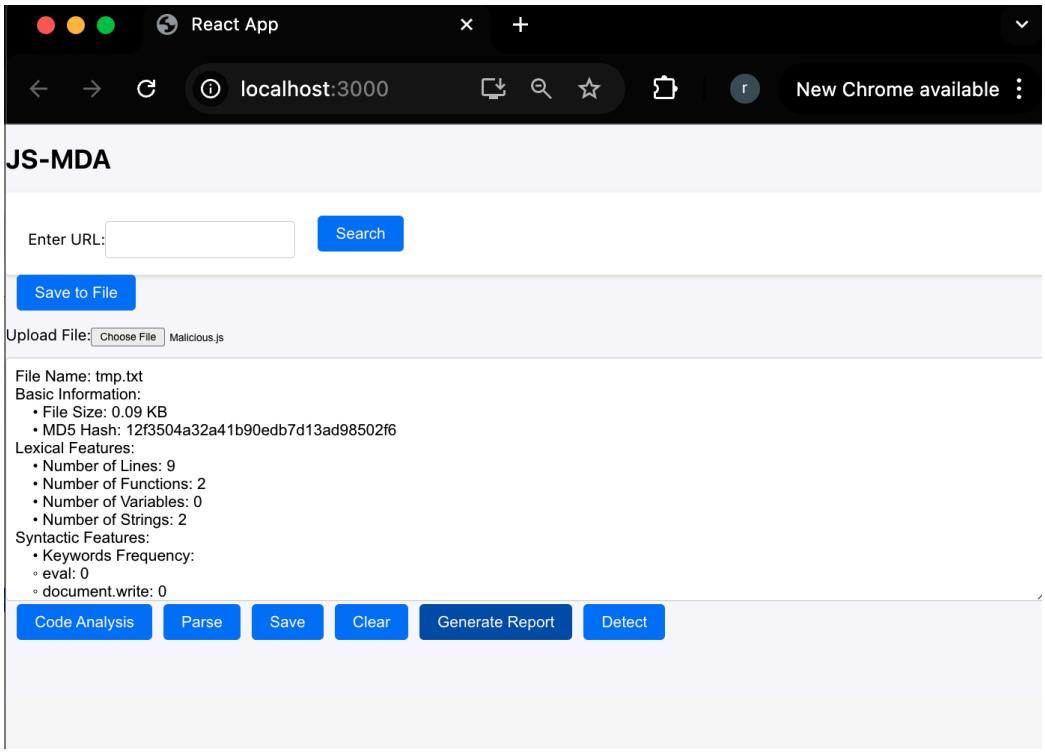


Figure 76: Static Report of the file

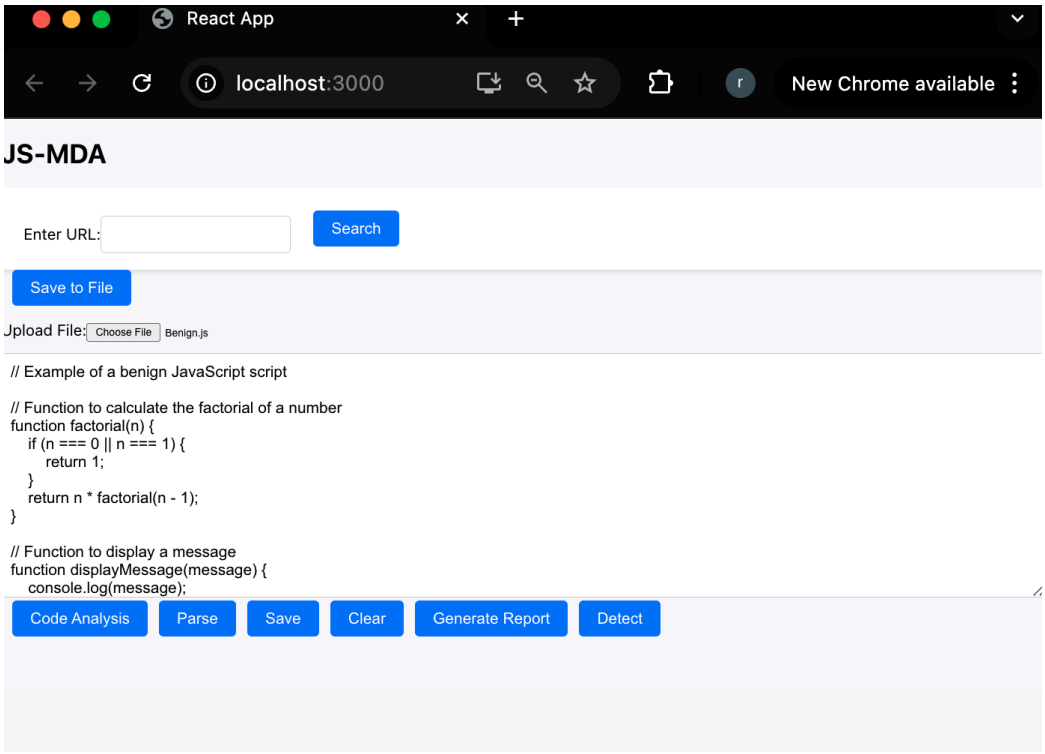
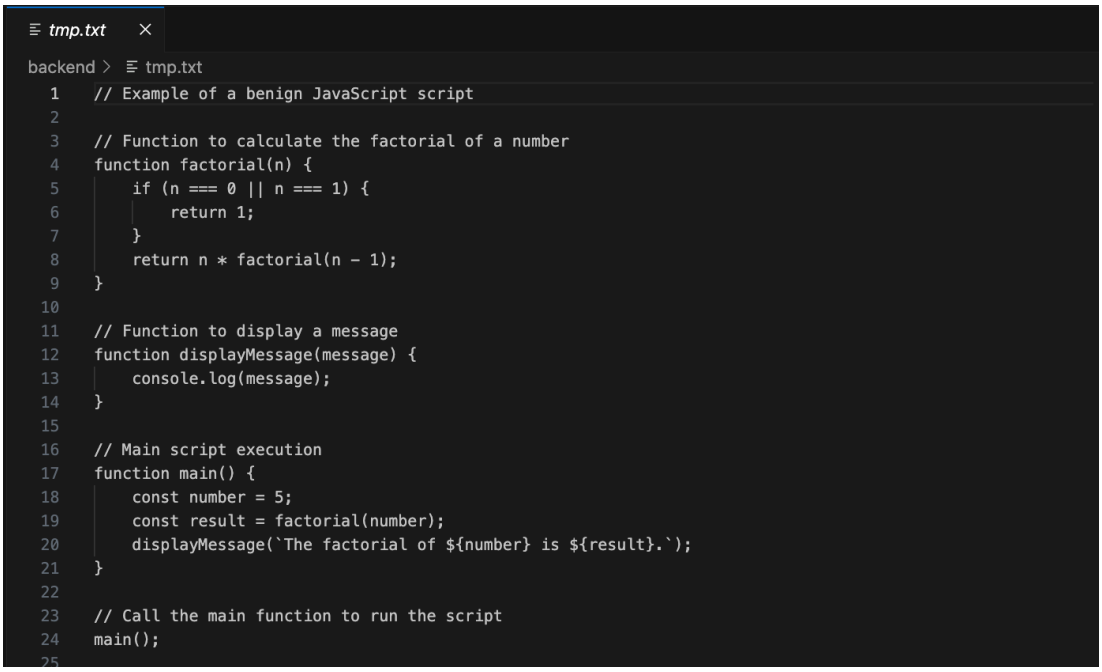


Figure 77: Loading benign javascript

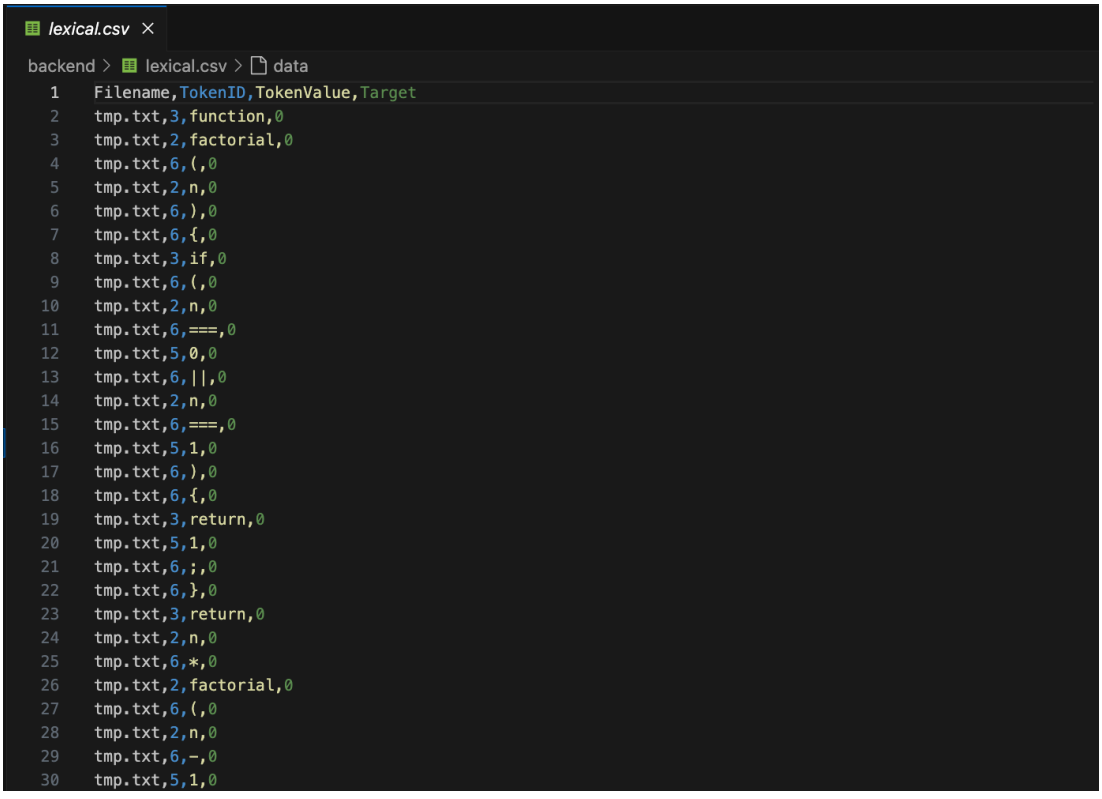


tmp.txt

backend > tmp.txt

```
// Example of a benign JavaScript script

// Function to calculate the factorial of a number
function factorial(n) {
    if (n === 0 || n === 1) {
        return 1;
    }
    return n * factorial(n - 1);
}

// Function to display a message
function displayMessage(message) {
    console.log(message);
}

// Main script execution
function main() {
    const number = 5;
    const result = factorial(number);
    displayMessage(`The factorial of ${number} is ${result}.`);
}

// Call the main function to run the script
main();
```

Figure 78: javascript code



lexical.csv

backend > lexical.csv > data

```
Filename,TokenID,TokenValue,Target
tmp.txt,3,function,0
tmp.txt,2,factorial,0
tmp.txt,6,(,0
tmp.txt,2,n,0
tmp.txt,6,),0
tmp.txt,6,{,0
tmp.txt,3,if,0
tmp.txt,6,(,0
tmp.txt,2,n,0
tmp.txt,6,===,0
tmp.txt,5,0,0
tmp.txt,6,||,0
tmp.txt,2,n,0
tmp.txt,6,===,0
tmp.txt,5,1,0
tmp.txt,6,),0
tmp.txt,6,{,0
tmp.txt,3,return,0
tmp.txt,5,1,0
tmp.txt,6,;,0
tmp.txt,6,},0
tmp.txt,3,return,0
tmp.txt,2,n,0
tmp.txt,6,*,0
tmp.txt,2,factorial,0
tmp.txt,6,(,0
tmp.txt,2,n,0
tmp.txt,6,-,0
tmp.txt,5,1,0
```

Figure 79: Tokenize data

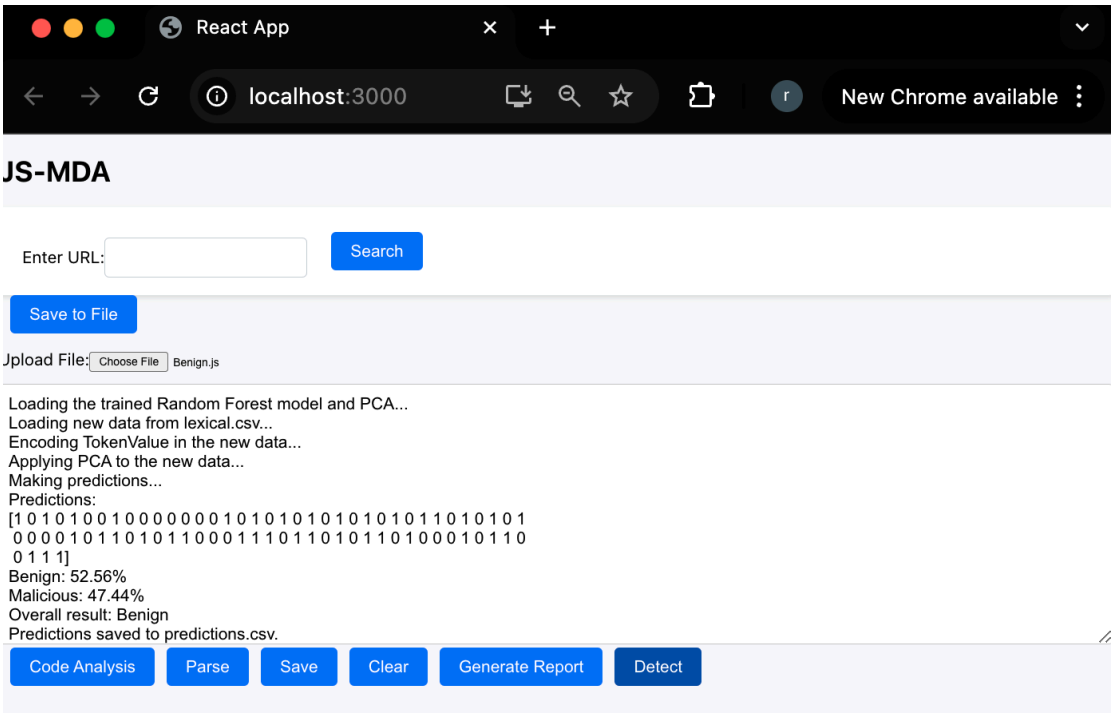


Figure 80: Classifier predection as Benign

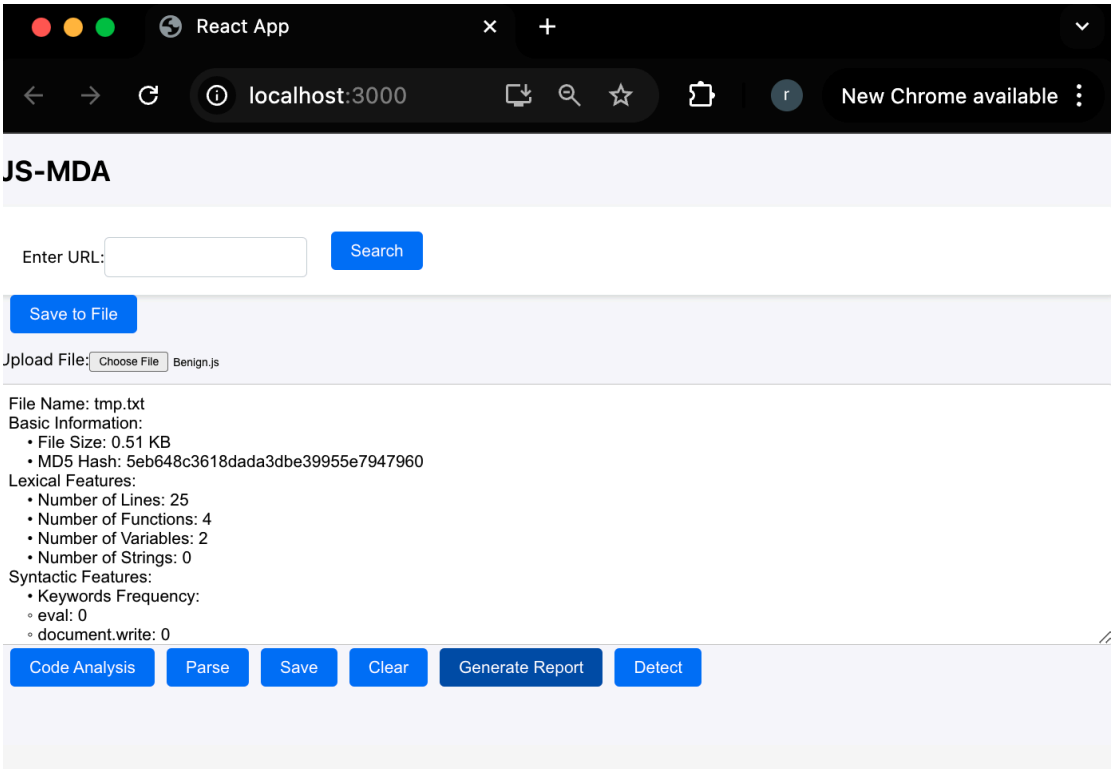


Figure 81: Static report

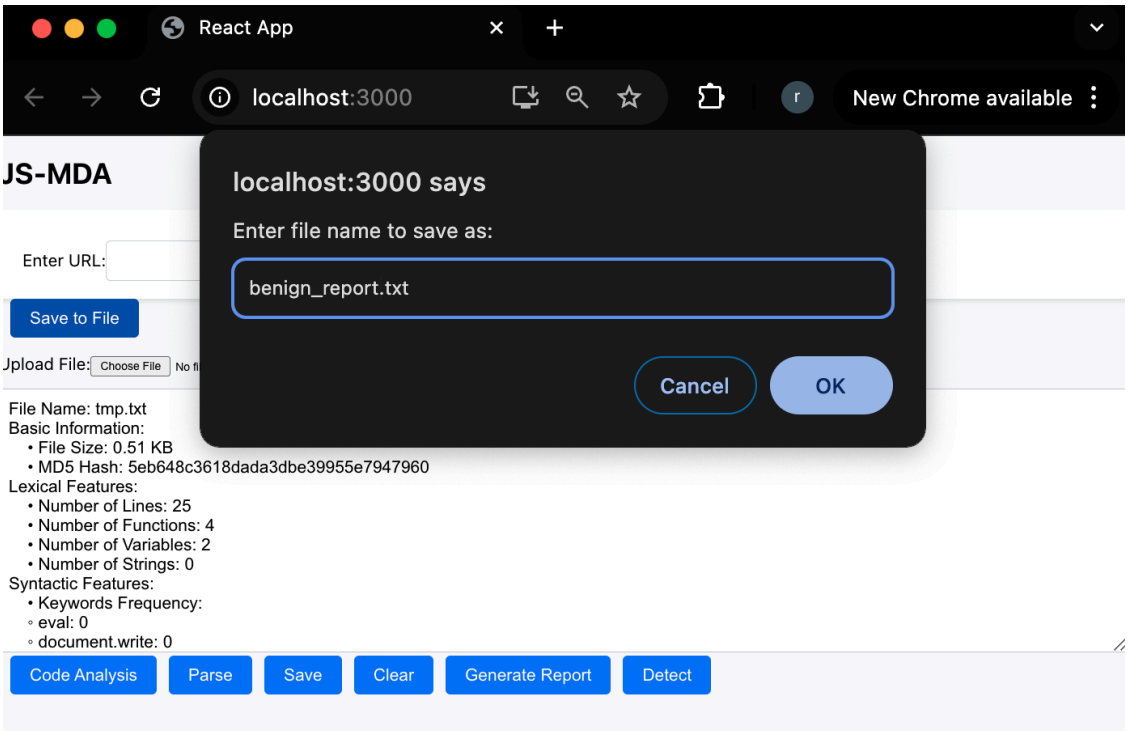


Figure 82: Saving file

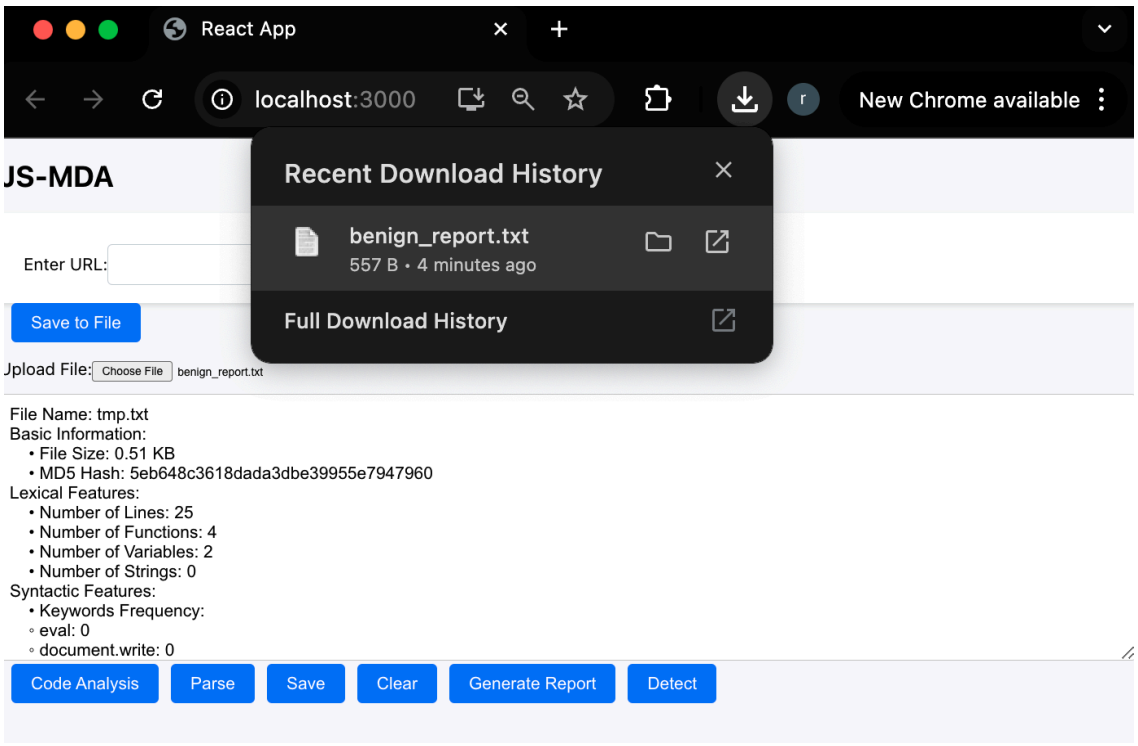


Figure 83: load the saved file benign_report.txt

detect_obfuscation.py — Machine_Learning

detect_obfuscation.py

```python
import re

def detect_obfuscation(js_code):
    # Patterns for common obfuscation methods
    hex_pattern = re.compile(r'\\x[0-9a-fA-F]{2}')
    unicode_pattern = re.compile(r'\\u[0-9a-fA-F]{4}')
    string_manipulation_pattern = re.compile(r'["\']\[\w+\]')
    base64_pattern = re.compile(r'([A-Za-z0-9+/]{4})*([A-Za-z0-9+/]{2}==|[A-Za-z0-9+/]{3}=)?')
    eval_pattern = re.compile(r'\beval\s*\(')
    char_code_pattern = re.compile(r'String\.fromCharCode')

    # More sophisticated patterns
    long_strings_pattern = re.compile(r'["\'][^"\']{20,}["\']')  # Very long strings
    excessive_concatenation_pattern = re.compile(r'(?:\+[^+]){5,}')  # Excessive concatenation
    obfuscated_function_pattern = re.compile(r'function\s*\(\w+\)\s*\{')
    excessive_obfuscation_pattern = re.compile(r'(?:\\x[0-9a-fA-F]{2}|\\u[0-9a-fA-F]{4}|fromCharCode|\beval\s*\(){20,}')

    # Check for presence of patterns
    hex_match = re.search(hex_pattern, js_code)
    unicode_match = re.search(unicode_pattern, js_code)
    string_manipulation_match = re.search(string_manipulation_pattern, js_code)
    base64_match = re.search(base64_pattern, js_code)
    eval_match = re.search(eval_pattern, js_code)
    char_code_match = re.search(char_code_pattern, js_code)
    long_strings_match = re.search(long_strings_pattern, js_code)
    excessive_concatenation_match = re.search(excessive_concatenation_pattern, js_code)
    obfuscated_function_match = re.search(obfuscated_function_pattern, js_code)
    excessive_obfuscation_match = re.search(excessive_obfuscation_pattern, js_code)

    obfuscation_patterns = [
        hex_match, unicode_match, string_manipulation_match, base64_match,
        eval_match, char_code_match, long_strings_match,
        excessive_concatenation_match, obfuscated_function_match,
        excessive_obfuscation_match
    ]

    obfuscation_score = sum(1 for match in obfuscation_patterns if match)

    # Determine if code is obfuscated based on the number of detected patterns
    if obfuscation_score > 3:  # Adjust threshold as needed
        return True
    return False

def detect_base64_encoded_from_file(filename):
    try:
        with open(filename, 'r') as file:
            txt = file.read().strip()

        # Check if the content is Base64 encoded
        base64_pattern = re.compile(r'^([A-Za-z0-9+/]{4})*([A-Za-z0-9+/]{3}=|[A-Za-z0-9+/]{2}==)?$')
        if re.search(base64_pattern, txt):
            return "Encoded"
        else:
            return "Non encoded"

    except FileNotFoundError:
        return f"Error: File '{filename}' not found."
    except Exception as e:
        return f"An error occurred: {e}"


def main():
    try:
        with open('tmp.txt', 'r') as file:
            js_code = file.read().strip()
        is_obfuscated = detect_obfuscation(js_code)

        if is_obfuscated:
```

Figure 84: Obfuscation identification script

detect_obfuscation.py — Machine_Learning

detect_obfuscation.py

```
    obfuscated_function_pattern = re.compile(r'function\s*\(\w+\)\s*\{')
    excessive_obfuscation_pattern = re.compile(r'(?:\\x[0-9a-fA-F]{2}|\\u[0-9a-fA-F]{4}|fromCharCode|\beval\s*\().{20,}')

    # Check for presence of patterns
    hex_match = re.search(hex_pattern, js_code)
    unicode_match = re.search(unicode_pattern, js_code)
    string_manipulation_match = re.search(string_manipulation_pattern, js_code)
    base64_match = re.search(base64_pattern, js_code)
    eval_match = re.search(eval_pattern, js_code)
    char_code_match = re.search(char_code_pattern, js_code)
    long_strings_match = re.search(long_strings_pattern, js_code)
    excessive_concatenation_match = re.search(excessive_concatenation_pattern, js_code)
    obfuscated_function_match = re.search(obfuscated_function_pattern, js_code)
    excessive_obfuscation_match = re.search(excessive_obfuscation_pattern, js_code)

    obfuscation_patterns = [
        hex_match, unicode_match, string_manipulation_match, base64_match,
        eval_match, char_code_match, long_strings_match,
        excessive_concatenation_match, obfuscated_function_match,
        excessive_obfuscation_match
    ]

    obfuscation_score = sum(1 for match in obfuscation_patterns if match)

    # Determine if code is obfuscated based on the number of detected patterns
    if obfuscation_score > 3:  # Adjust threshold as needed
        return True
    return False

def detect_base64_encoded_from_file(filename):
    try:
        with open(filename, 'r') as file:
            txt = file.read().strip()

        # Check if the content is Base64 encoded
        base64_pattern = re.compile(r'^([A-Za-z0-9+/]{4})*([A-Za-z0-9+/]{3}=|[A-Za-z0-9+/]{2}==)?$')
        if re.search(base64_pattern, txt):
            return "Encoded"
        else:
            return "Non encoded"

    except FileNotFoundError:
        return f"Error: File '{filename}' not found."
    except Exception as e:
        return f"An error occurred: {e}"


def main():
    try:
        with open('tmp.txt', 'r') as file:
            js_code = file.read().strip()
        is_obfuscated = detect_obfuscation(js_code)

        if is_obfuscated:
            print("The JavaScript code appears to be obfuscated.")
        else:
            print("The JavaScript code does not appear to be obfuscated.")

        base64_status = detect_base64_encoded_from_file('tmp.txt')
        print(f"Base64 status:{base64_status}.")
    except FileNotFoundError:
        print(f"Error: File 'tmp.txt' not found.")
    except Exception as e:
        print(f"An error occurred: {e}")

if __name__ == "__main__":
    main()
```

Figure 84.1: Encryption identification script

static_report.py — Machine_Learning

static_report.py

```python
import csv
import argparse
import hashlib
import os

def calculate_md5(file_path):
    with open(file_path, 'rb') as file:
        file_hash = hashlib.md5()
        while chunk := file.read(8192):
            file_hash.update(chunk)
        return file_hash.hexdigest()

def analyze_js_file(file_path):
    with open(file_path, 'r') as file:
        content = file.read()
        num_lines = content.count('\n') + 1
        num_functions = content.count('function')
        num_variables = content.count('var') + content.count('let') + content.count('const')
        num_strings = content.count('"') + content.count("'") // 2
        keywords = {
            'eval': content.count('eval'),
            'document.write': content.count('document.write'),
            'setTimeout': content.count('setTimeout'),
            'setInterval': content.count('setInterval'),
            'escape': content.count('escape'),
        }
        suspicious_patterns = {
            'Obfuscation Detected': 'Yes' if 'eval' in content or 'escape' in content else 'No',
            'Dynamic Code Execution': 'Yes' if 'document.write' in content else 'No',
            'Network Requests': 'Yes' if 'iframe' in content else 'No',
            'Cookie Access': 'Yes' if 'document.cookie' in content else 'No'
        }
        return {
            'num_lines': num_lines,
            'num_functions': num_functions,
            'num_variables': num_variables,
            'num_strings': num_strings,
            'keywords': keywords,
            'suspicious_patterns': suspicious_patterns
        }

def generate_report(file_path):
    file_size = os.path.getsize(file_path) / 1024  # File size in KB
    md5_hash = calculate_md5(file_path)
    analysis = analyze_js_file(file_path)

    report = f"""File Name: {os.path.basename(file_path)}
Basic Information:
    • File Size: {file_size:.2f} KB
    • MD5 Hash: {md5_hash}
Lexical Features:
    • Number of Lines: {analysis['num_lines']}
    • Number of Functions: {analysis['num_functions']}
    • Number of Variables: {analysis['num_variables']}
    • Number of Strings: {analysis['num_strings']}
Syntactic Features:
    • Keywords Frequency:"""

    for keyword, count in analysis['keywords'].items():
        report += f"\n    ◦ {keyword}: {count}"

    report += "\n    • Suspicious Patterns:"

    for pattern, detected in analysis['suspicious_patterns'].items():
        report += f"\n    ◦ {pattern}: {detected}"

    return report
```

Figure 85: Static analysis report script

Figure 86.1: Static analysis report script

```
127.0.0.1 - - [17/Aug/2024 16:44:33] "POST /api/dete
ct HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2024 16:45:48] "OPTIONS /api/p
arse HTTP/1.1" 200 -
Received content: //malicious script
function l3() {
    return 'ect';
}

function l4() {
    return ' fn';
}

127.0.0.1 - - [17/Aug/2024 16:45:49] "POST /api/pars
e HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2024 16:47:22] "OPTIONS /api/r
eport HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2024 16:47:22] "POST /api/repo
rt HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2024 16:47:39] "OPTIONS /api/r
eport HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2024 16:47:39] "POST /api/repo
rt HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2024 16:52:13] "OPTIONS /api/s
```

Figure 87: API communication (request and response)

```
ToTmp HTTP/1.1" 200 -
127.0.0.1 - - [17/Aug/2024 16:52:57] "OPTIONS /api/p
arse HTTP/1.1" 200 -
Received content: // Example of a benign JavaScript
script

// Function to calculate the factorial of a number
function factorial(n) {
    if (n === 0 || n === 1) {
        return 1;
    }
    return n * factorial(n - 1);
}

// Function to display a message
function displayMessage(message) {
    console.log(message);
}

// Main script execution
function main() {
    const number = 5;
    const result = factorial(number);
    displayMessage(`The factorial of ${number} is ${
result}.`);
}

// Call the main function to run the script
main();
```

Figure 88: API communication (backend , flask and python scripts)

Machine learning

Figure 89: Parsed lexical and syntactic dataset in CSV format

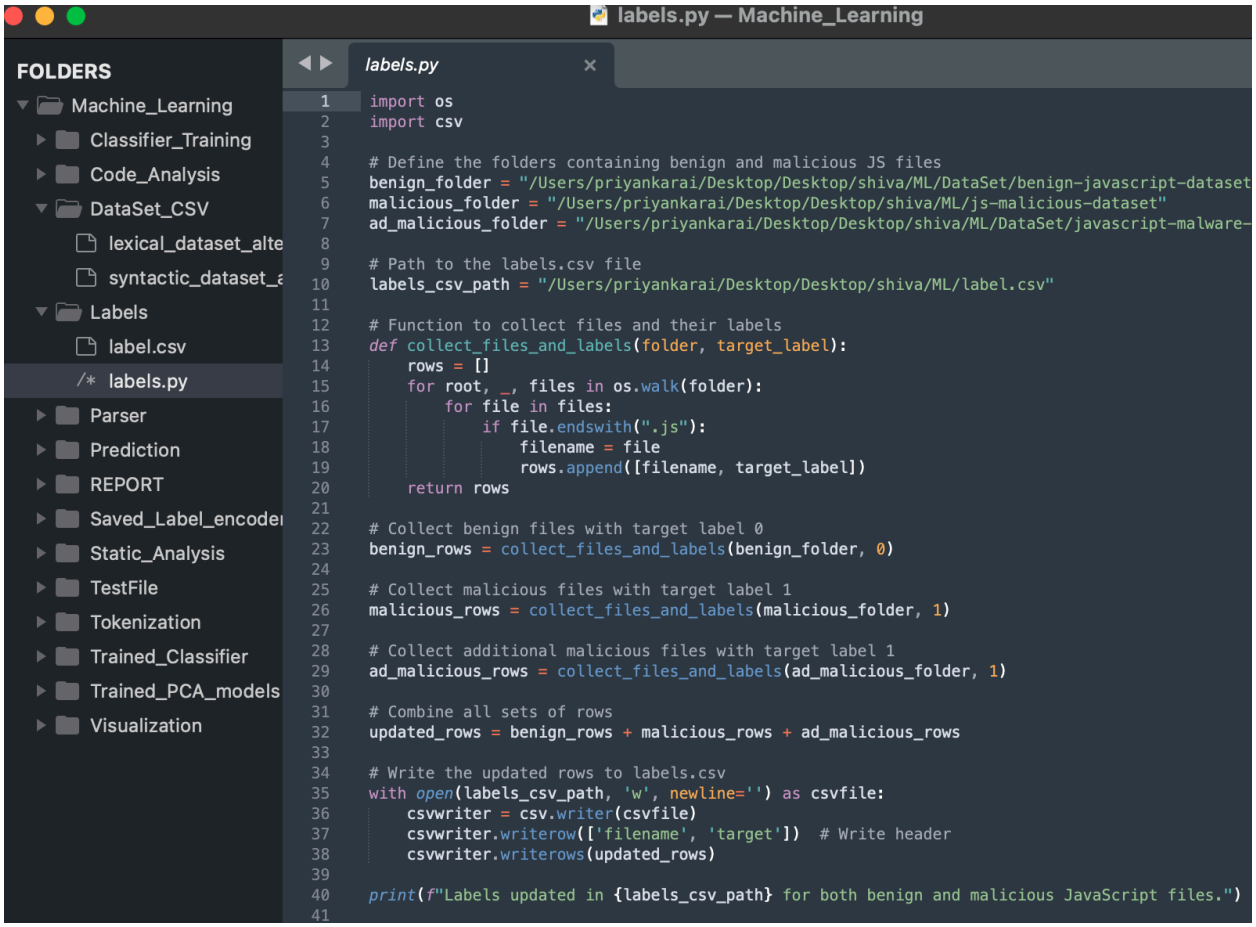


Figure 90: Labels.py

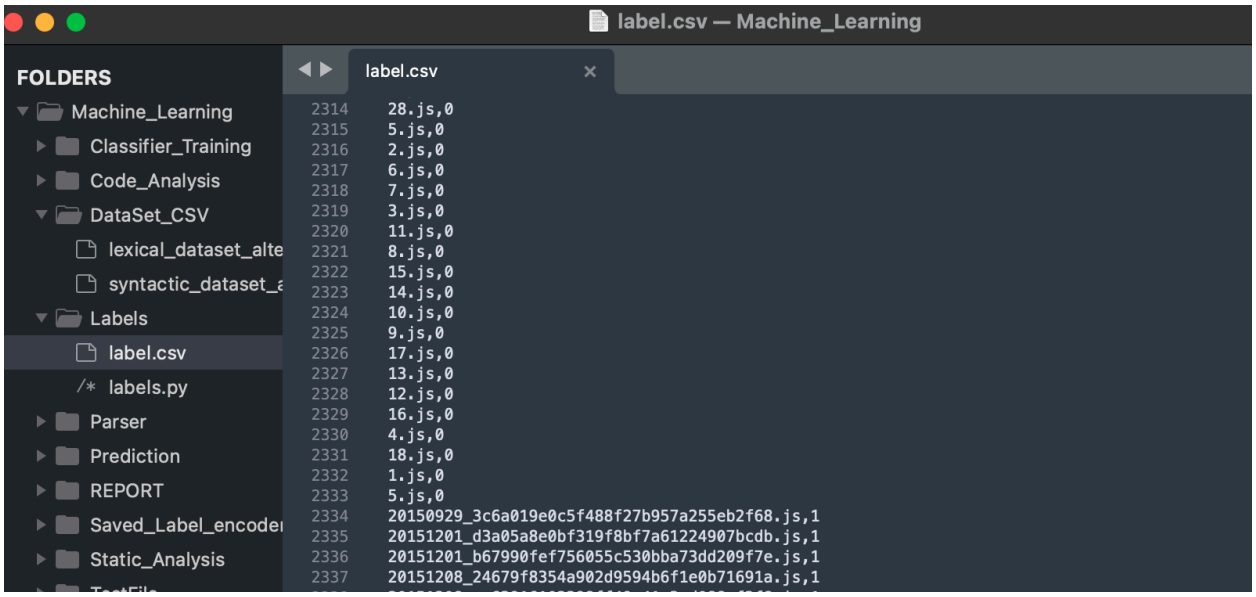


Figure 91: label.csv

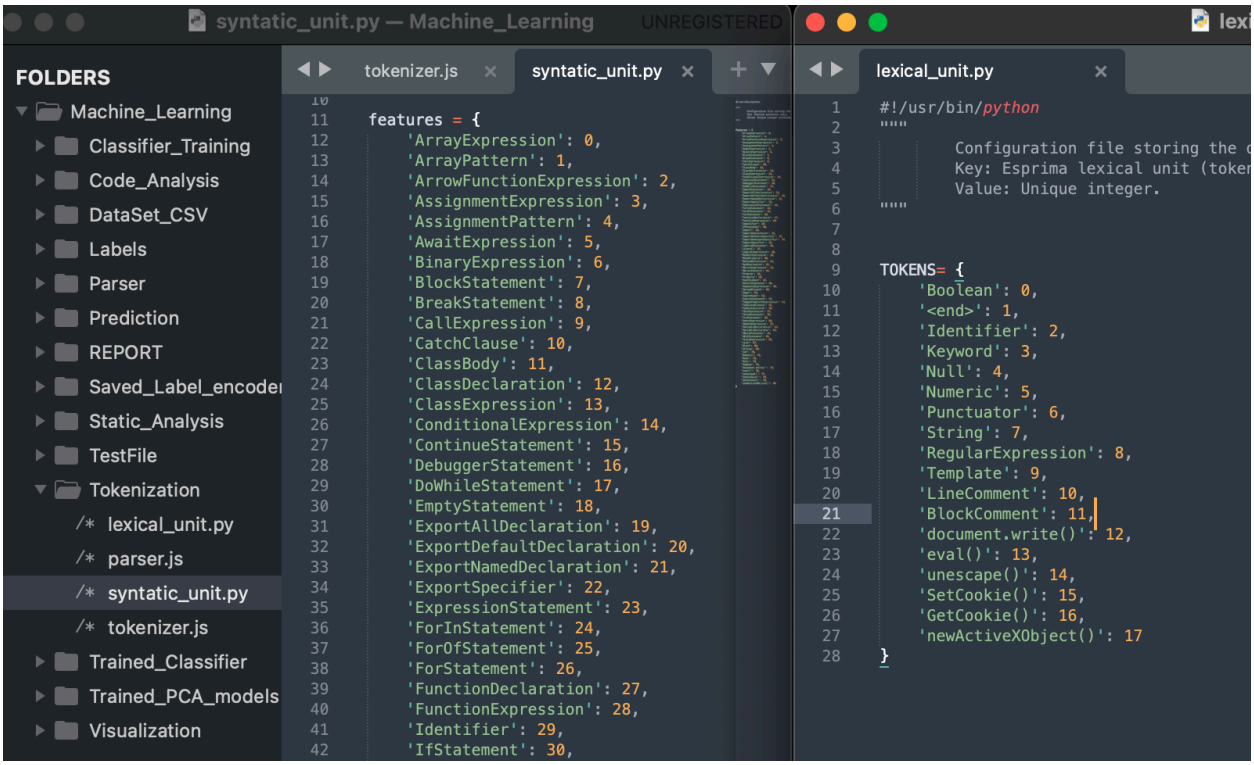


Figure 92: Manual mapping of Tokens and Features

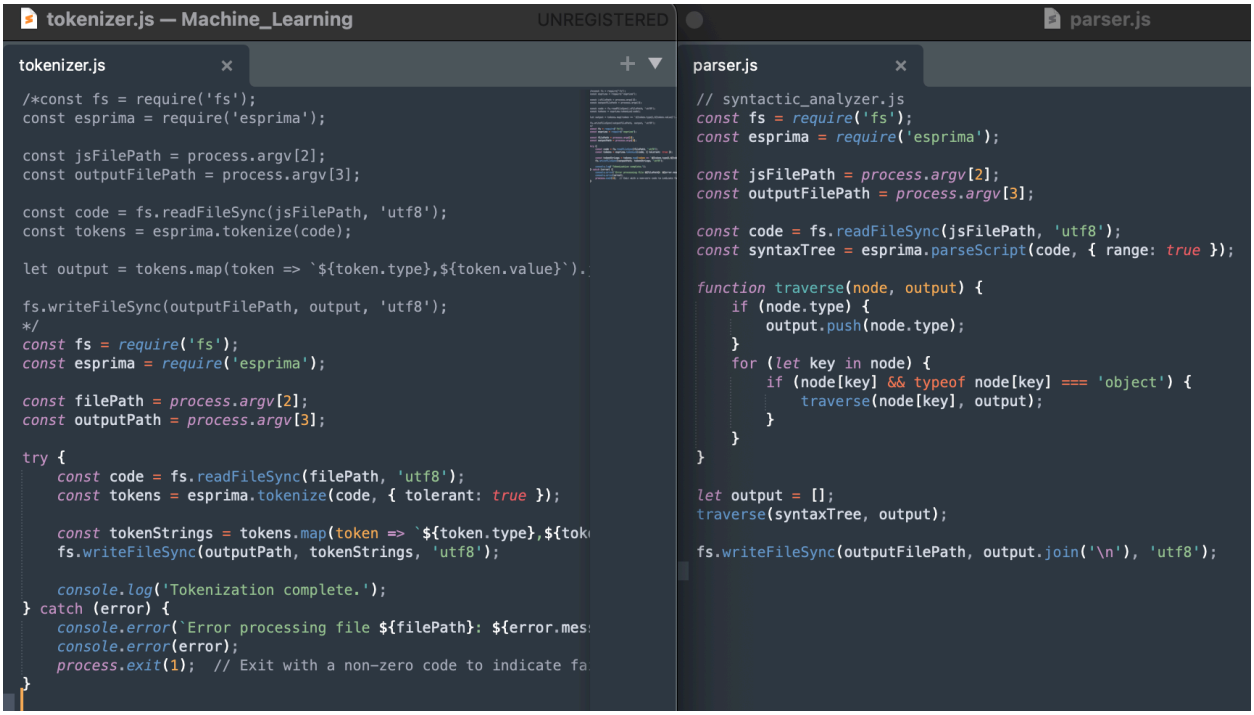


Figure 93: code to tokenize and feature

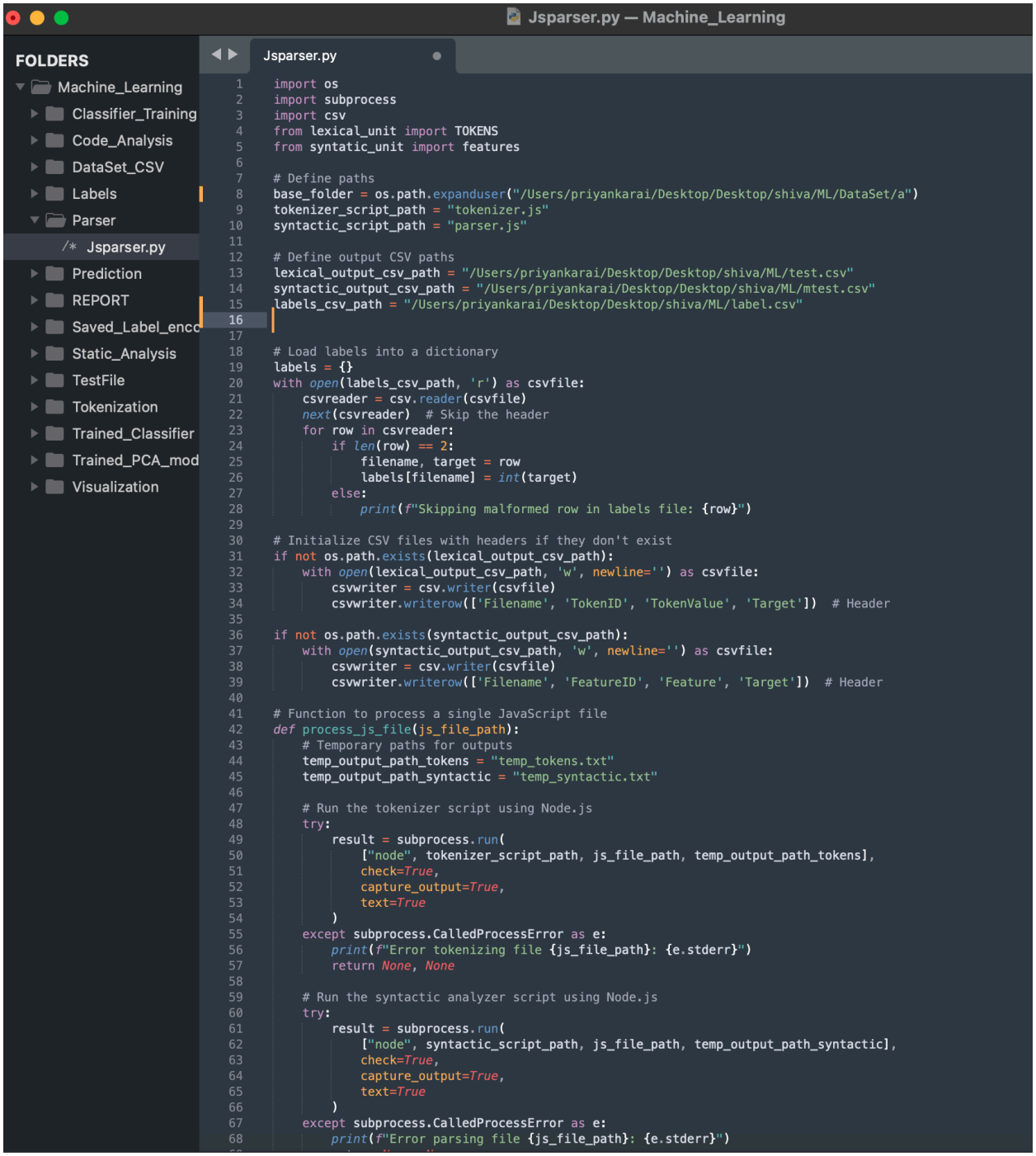


Figure 94: Parser

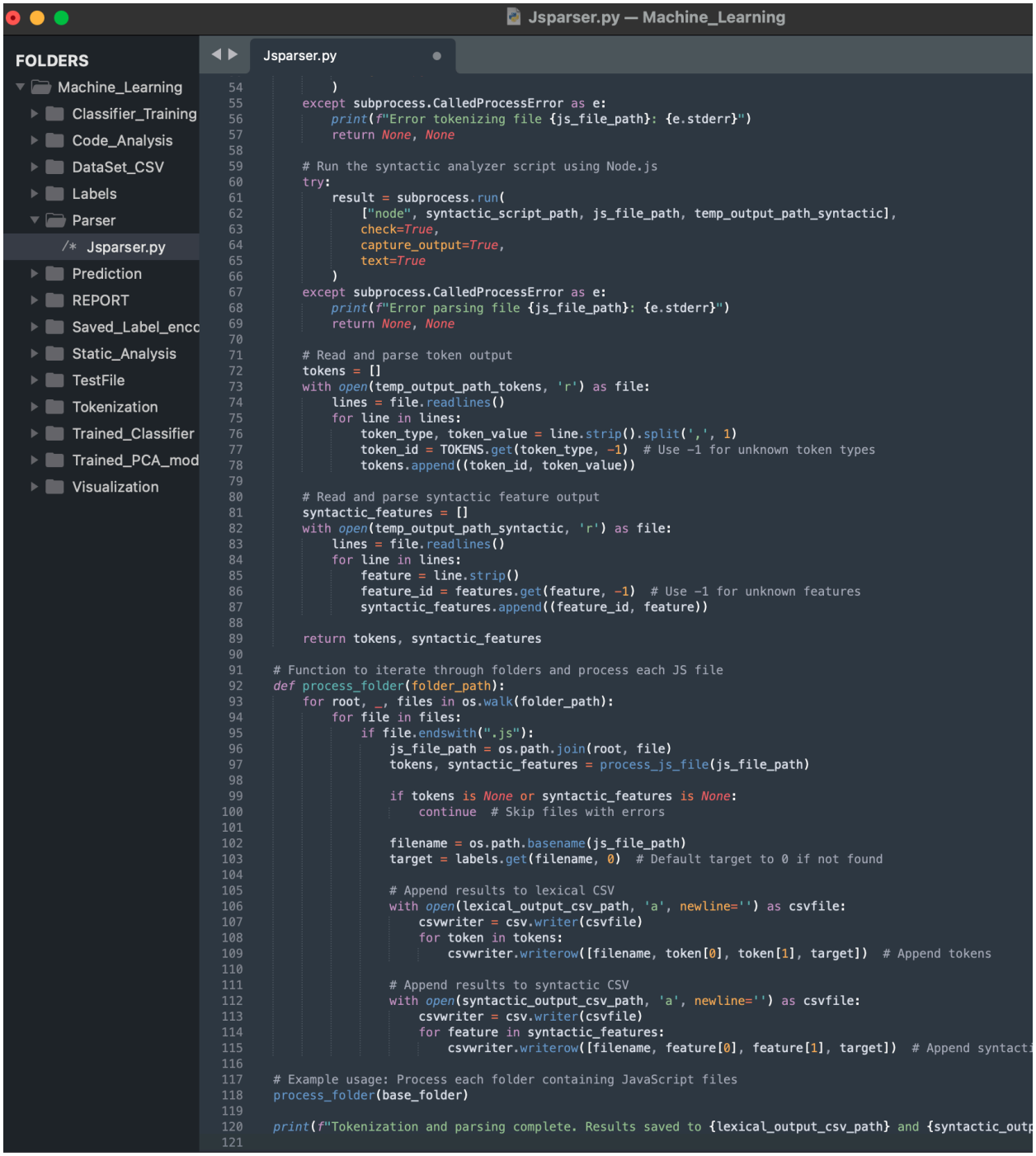


Figure 94.1: Parser

RFM.py — Machine_Learning

RFM.py

```python
import pandas as pd
from sklearn.preprocessing import LabelEncoder
from sklearn.decomposition import PCA
import joblib
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report
from imblearn.over_sampling import SMOTE
import time

# Function to preprocess categorical columns
def preprocess_categorical_columns(df, categorical_columns):
    print(f"Preprocessing columns: {categorical_columns}")
    label_encoders = {}
    for col in categorical_columns:
        if col in df.columns:
            le = LabelEncoder()
            df[col] = le.fit_transform(df[col].astype(str))  # Ensure column is treated as string
            label_encoders[col] = le
            print(f"Encoded {col}: {df[col].head()}")
        else:
            print(f"Column '{col}' not found in DataFrame.")
    return df, label_encoders

# Define paths to your CSV files
Lexical_DataSet_csv = "lexical_dataset_alternate.csv"
Syntactic_DataSet_csv = "syntactic_dataset_alternate.csv"

# Load data into DataFrames with explicit dtype specification
start_time = time.time()
df_lexical = pd.read_csv(Lexical_DataSet_csv, dtype={'Filename': str, 'TokenID': int, 'TokenValue': str, 'Target': int})
df_syntactic = pd.read_csv(Syntactic_DataSet_csv, dtype={'Filename': str, 'FeatureID': int, 'Feature': str, 'Target': int})
end_time = time.time()
print(f"Datasets loaded in {end_time - start_time:.2f} seconds.")

# Debug: Print columns present in the DataFrames
print("Lexical DataFrame columns:", df_lexical.columns)
print("Syntactic DataFrame columns:", df_syntactic.columns)

# Checking columns
expected_columns_lexical = ['Filename', 'TokenID', 'TokenValue', 'Target']
missing_columns_lexical = [col for col in expected_columns_lexical if col not in df_lexical.columns]

expected_columns_syntactic = ['Filename', 'FeatureID', 'Feature', 'Target']
missing_columns_syntactic = [col for col in expected_columns_syntactic if col not in df_syntactic.columns]

if missing_columns_lexical:
    print(f"Missing columns in lexical DataFrame: {missing_columns_lexical}")
if missing_columns_syntactic:
    print(f"Missing columns in syntactic DataFrame: {missing_columns_syntactic}")

# Preprocess lexical data
def preprocess_lexical_data(df):
    df['TokenID'].fillna(0, inplace=True)
    df['TokenValue'].fillna('', inplace=True)
    df['TokenID'] = df['TokenID'].astype(int)
    df, label_encoders = preprocess_categorical_columns(df, ['TokenValue'])  # Adjust based on actual columns
    return df, label_encoders

# Preprocess syntactic data
def preprocess_syntactic_data(df):
    df['FeatureID'].fillna(0, inplace=True)
    df['Feature'].fillna('', inplace=True)
    df['FeatureID'] = df['FeatureID'].astype(int)
    df, label_encoders = preprocess_categorical_columns(df, ['Feature'])  # Adjust based on actual columns
    return df, label_encoders

# Preprocess both DataFrames
```

Figure 95: Random Forest Training Code

RFM.py — Machine_Learning

RFM.py

```
# Preprocess both DataFrames
start_time = time.time()
df_lexical, lexical_encoders = preprocess_lexical_data(df_lexical)
end_time = time.time()
print(f"Lexical DataFrame preprocessed in {end_time - start_time:.2f} seconds.")

start_time = time.time()
df_syntactic, syntactic_encoders = preprocess_syntactic_data(df_syntactic)
end_time = time.time()
print(f"Syntactic DataFrame preprocessed in {end_time - start_time:.2f} seconds.")

# Save entire label encoders dictionary
joblib.dump(lexical_encoders['TokenValue'], 'label_encoder_TokenValues.pkl')
joblib.dump(syntactic_encoders['Feature'], 'label_encoder_Features.pkl')
print("Label encoders saved.")

# Separate features and target for lexical dataset
X_lexical = df_lexical.drop(columns=['Target', 'Filename', 'TokenID'])
y_lexical = df_lexical['Target']
print("Lexical features and target separated.")

# Separate features and target for syntactic dataset
X_syntactic = df_syntactic.drop(columns=['Target', 'Filename', 'FeatureID'])
y_syntactic = df_syntactic['Target']
print("Syntactic features and target separated.")

# Determine n_components dynamically based on data size
n_components_lexical = min(50, X_lexical.shape[1])  # Ensure it's within valid range
n_components_syntactic = min(50, X_syntactic.shape[1])  # Ensure it's within valid range
print(f"PCA components: Lexical = {n_components_lexical}, Syntactic = {n_components_syntactic}")

# Apply PCA to reduce dimensionality
start_time = time.time()
pca_lexical = PCA(n_components=n_components_lexical)
X_lexical_pca = pca_lexical.fit_transform(X_lexical)
end_time = time.time()
print(f"PCA applied to lexical dataset in {end_time - start_time:.2f} seconds.")

start_time = time.time()
pca_syntactic = PCA(n_components=n_components_syntactic)
X_syntactic_pca = pca_syntactic.fit_transform(X_syntactic)
end_time = time.time()
print(f"PCA applied to syntactic dataset in {end_time - start_time:.2f} seconds.")

# Save PCA models
joblib.dump(pca_lexical, 'pca_lexical.pkl')
joblib.dump(pca_syntactic, 'pca_syntactic.pkl')
print("PCA models saved.")

# Split data into training and testing sets for both datasets
start_time = time.time()
X_lexical_train, X_lexical_test, y_lexical_train, y_lexical_test = train_test_split(X_lexical_pca, y_lexical, test_size=0.2, random_
X_syntactic_train, X_syntactic_test, y_syntactic_train, y_syntactic_test = train_test_split(X_syntactic_pca, y_syntactic, test_size=
end_time = time.time()
print(f"Data split into training and testing sets in {end_time - start_time:.2f} seconds.")

# Apply SMOTE to handle class imbalance in lexical dataset
start_time = time.time()
print("Applying SMOTE to lexical dataset...")
smote = SMOTE(random_state=42)
X_lexical_train_res, y_lexical_train_res = smote.fit_resample(X_lexical_train, y_lexical_train)
end_time = time.time()
print(f"SMOTE applied to lexical dataset in {end_time - start_time:.2f} seconds.")

# Apply SMOTE to handle class imbalance in syntactic dataset
start_time = time.time()
print("Applying SMOTE to syntactic dataset...")
X_syntactic_train_res, y_syntactic_train_res = smote.fit_resample(X_syntactic_train, y_syntactic_train)
```

Figure 95.1: Random Forest Training Code

```python
# Apply SMOTE to handle class imbalance in syntactic dataset
start_time = time.time()
print("Applying SMOTE to syntactic dataset...")
X_syntactic_train_res, y_syntactic_train_res = smote.fit_resample(X_syntactic_train, y_syntactic_train)
end_time = time.time()
print(f"SMOTE applied to syntactic dataset in {end_time - start_time:.2f} seconds.")

# Initialize Random Forest classifiers
print("Initializing Random Forest classifiers...")
clf_lexical_rf = RandomForestClassifier(random_state=42)
clf_syntactic_rf = RandomForestClassifier(random_state=42)

# Fit Random Forest classifiers on training data
start_time = time.time()
print("Fitting Random Forest on lexical dataset...")
clf_lexical_rf.fit(X_lexical_train_res, y_lexical_train_res)
end_time = time.time()
print(f"Random Forest trained on lexical dataset in {end_time - start_time:.2f} seconds.")

start_time = time.time()
print("Fitting Random Forest on syntactic dataset...")
clf_syntactic_rf.fit(X_syntactic_train_res, y_syntactic_train_res)
end_time = time.time()
print(f"Random Forest trained on syntactic dataset in {end_time - start_time:.2f} seconds.")

# Save the trained models
joblib.dump(clf_lexical_rf, 'random_forest_lexical.pkl')
joblib.dump(clf_syntactic_rf, 'random_forest_syntactic.pkl')
print("Random Forest models saved.")

# Predict on test data with Random Forest classifiers
start_time = time.time()
print("Predicting on lexical test data with Random Forest...")
y_lexical_rf_pred = clf_lexical_rf.predict(X_lexical_test)
end_time = time.time()
print(f"Prediction on lexical test data completed in {end_time - start_time:.2f} seconds.")

start_time = time.time()
print("Predicting on syntactic test data with Random Forest...")
y_syntactic_rf_pred = clf_syntactic_rf.predict(X_syntactic_test)
end_time = time.time()
print(f"Prediction on syntactic test data completed in {end_time - start_time:.2f} seconds.")

# Evaluate Random Forest classifiers
print("\nLexical Dataset Classification Report (Random Forest):")
print(classification_report(y_lexical_test, y_lexical_rf_pred, digits=4))

print("\nSyntactic Dataset Classification Report (Random Forest):")
print(classification_report(y_syntactic_test, y_syntactic_rf_pred, digits=4))

```

Figure 95.2: Random Forest Training Code

Figure 96: Logistic Regression model training code

```
    # Apply PCA
    pca_lexical = PCA(n_components=min(50, X_lexical.shape[1]))
    X_lexical_pca = pca_lexical.fit_transform(X_lexical)
    joblib.dump(pca_lexical, 'pca_lexical.pkl')
    logging.info(f"PCA applied to lexical data. Explained variance ratio: {pca_lexical.explained_variance_ratio_}")

    pca_syntactic = PCA(n_components=min(50, X_syntactic.shape[1]))
    X_syntactic_pca = pca_syntactic.fit_transform(X_syntactic)
    joblib.dump(pca_syntactic, 'pca_syntactic.pkl')
    logging.info(f"PCA applied to syntactic data. Explained variance ratio: {pca_syntactic.explained_variance_ratio_}")

    # Split data
    X_lexical_train, X_lexical_test, y_lexical_train, y_lexical_test = train_test_split(X_lexical_pca, y_lexical, test_size=0.2, random_
    X_syntactic_train, X_syntactic_test, y_syntactic_train, y_syntactic_test = train_test_split(X_syntactic_pca, y_syntactic, test_size=
    logging.info(f"Data split: Lexical train shape: {X_lexical_train.shape}, Syntactic train shape: {X_syntactic_train.shape}")

    # Apply SMOTE
    smote_lexical = SMOTE(random_state=42)
    X_lexical_train_res, y_lexical_train_res = smote_lexical.fit_resample(X_lexical_train, y_lexical_train)
    logging.debug(f"After SMOTE: Lexical train shape: {X_lexical_train_res.shape}")

    smote_syntactic = SMOTE(random_state=42)
    X_syntactic_train_res, y_syntactic_train_res = smote_syntactic.fit_resample(X_syntactic_train, y_syntactic_train)
    logging.debug(f"After SMOTE: Syntactic train shape: {X_syntactic_train_res.shape}")

    # Train and save Logistic Regression models
    clf_lexical_lr = LogisticRegression(random_state=42)
    clf_syntactic_lr = LogisticRegression(random_state=42)

    logging.info("Training Logistic Regression model on lexical data...")
    clf_lexical_lr.fit(X_lexical_train_res, y_lexical_train_res)
    logging.info("Training Logistic Regression model on syntactic data...")
    clf_syntactic_lr.fit(X_syntactic_train_res, y_syntactic_train_res)

    joblib.dump(clf_lexical_lr, 'lr_lexical.pkl')
    joblib.dump(clf_syntactic_lr, 'lr_syntactic.pkl')
    logging.info("Logistic Regression models saved.")

    # Predictions and evaluation
    y_lexical_lr_pred = clf_lexical_lr.predict(X_lexical_test)
    y_syntactic_lr_pred = clf_syntactic_lr.predict(X_syntactic_test)

    logging.info("Generating classification reports...")
    print("\nLexical Dataset Classification Report (Logistic Regression):")
    print(classification_report(y_lexical_test, y_lexical_lr_pred))

    print("\nSyntactic Dataset Classification Report (Logistic Regression):")
    print(classification_report(y_syntactic_test, y_syntactic_lr_pred))

```

Figure 96.1 Logistic Regression

RFM_prediction_lexical.py — Machine_Learning

RFM_prediction_lexical.py

```python
import pandas as pd
import joblib
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, accuracy_score
import seaborn as sns

# Load the trained Random Forest model and PCA
print("Loading the trained Random Forest model and PCA...")
lr_model = joblib.load('random_forest_lexical.pkl')
pca = joblib.load('pca_lexical.pkl')
token_value_encoder = joblib.load('label_encoder_TokenValues.pkl')

# Load the new lexical data (e.g., 'test.csv')
filename = 'test.csv'
dtype_spec = {
    'Filename': str,
    'TokenID': int,
    'TokenValue': str,
    'Target': int
}

print(f"Loading new data from {filename}...")
new_data = pd.read_csv(filename, dtype=dtype_spec)

# Encode the TokenValue using the same encoder
print("Encoding TokenValue in the new data...")
new_data['TokenValue'] = token_value_encoder.transform(new_data['TokenValue'])

# Prepare the feature set (X) - In this case, only TokenValue is used since PCA is applied to it
X_new = new_data[['TokenValue']]

# Apply PCA to the new data
print("Applying PCA to the new data...")
X_new_pca = pca.transform(X_new)

# Make predictions using the Random Forest model
print("Making predictions...")
predictions = lr_model.predict(X_new_pca)

# Convert predictions to [0, 1] format
print("Predictions:")
print(predictions)

# Calculate the percentage of "Malicious" and "Benign"
benign_count = (predictions == 0).sum()
malicious_count = (predictions == 1).sum()
total_count = len(predictions)

benign_percentage = (benign_count / total_count) * 100
malicious_percentage = (malicious_count / total_count) * 100

# Print percentages
print(f"Benign: {benign_percentage:.2f}%")
print(f"Malicious: {malicious_percentage:.2f}%")

# Determine overall result
overall_result = "Malicious" if malicious_count > benign_count else "Benign"
print(f"Overall result: {overall_result}")

# Save the predictions if needed
output_filename = 'predictions.csv'
new_data['Predicted'] = predictions
new_data.to_csv(output_filename, index=False)
print(f"Predictions saved to {output_filename}.")
```

Figure 97: RF model prediction code

LR_prediction_lexical.py — Machine_Learning

LR_prediction_lexical.py

```python
import pandas as pd
import joblib
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, accuracy_score
import seaborn as sns

# Load the trained Random Forest model and PCA
print("Loading the trained Random Forest model and PCA...")
lr_model = joblib.load('lr_lexical.pkl')
pca = joblib.load('pca_lexical.pkl')
token_value_encoder = joblib.load('label_encoder_TokenValues.pkl')

# Load the new lexical data (e.g., 'test.csv')
filename = 'test.csv'
dtype_spec = {
    'Filename': str,
    'TokenID': int,
    'TokenValue': str,
    'Target': int
}

print(f"Loading new data from {filename}...")
new_data = pd.read_csv(filename, dtype=dtype_spec)

# Encode the TokenValue using the same encoder
print("Encoding TokenValue in the new data...")
new_data['TokenValue'] = token_value_encoder.transform(new_data['TokenValue'])

# Prepare the feature set (X) - In this case, only TokenValue is used since PCA is applied to it
X_new = new_data[['TokenValue']]

# Apply PCA to the new data
print("Applying PCA to the new data...")
X_new_pca = pca.transform(X_new)

# Make predictions using the Random Forest model
print("Making predictions...")
predictions = lr_model.predict(X_new_pca)

# Convert predictions to [0, 1] format
print("Predictions:")
print(predictions)

# Calculate the percentage of "Malicious" and "Benign"
benign_count = (predictions == 0).sum()
malicious_count = (predictions == 1).sum()
total_count = len(predictions)

benign_percentage = (benign_count / total_count) * 100
malicious_percentage = (malicious_count / total_count) * 100

# Print percentages
print(f"Benign: {benign_percentage:.2f}%")
print(f"Malicious: {malicious_percentage:.2f}%")

# Determine overall result
overall_result = "Malicious" if malicious_count > benign_count else "Benign"
print(f"Overall result: {overall_result}")

# Save the predictions if needed
output_filename = 'predictions.csv'
new_data['Predicted'] = predictions
new_data.to_csv(output_filename, index=False)
print(f"Predictions saved to {output_filename}.")
```

Figure 98: LR prediction code

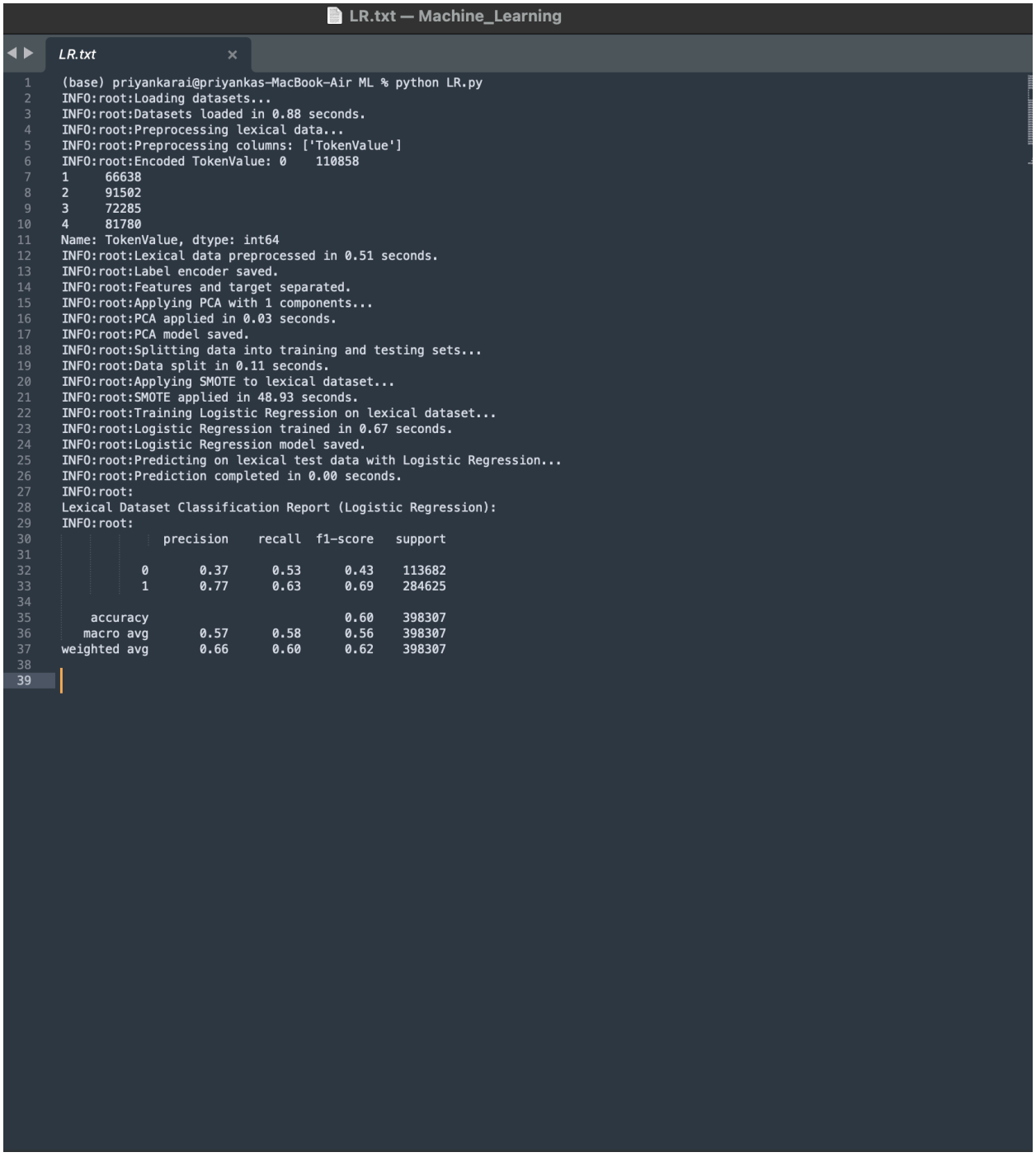


Figure 99: LR training and evaluation result

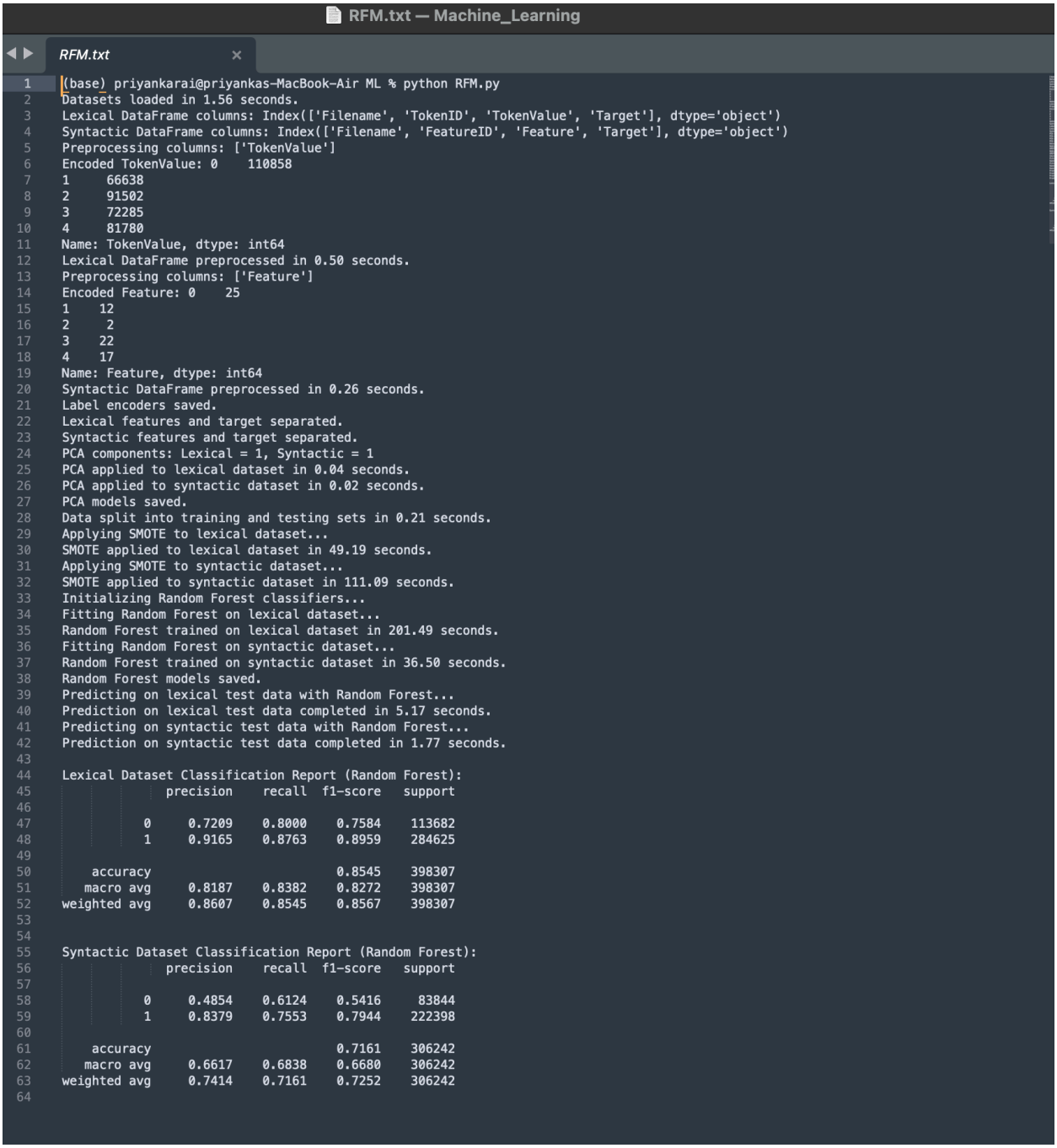


Figure 100: RF model training and evaluation result

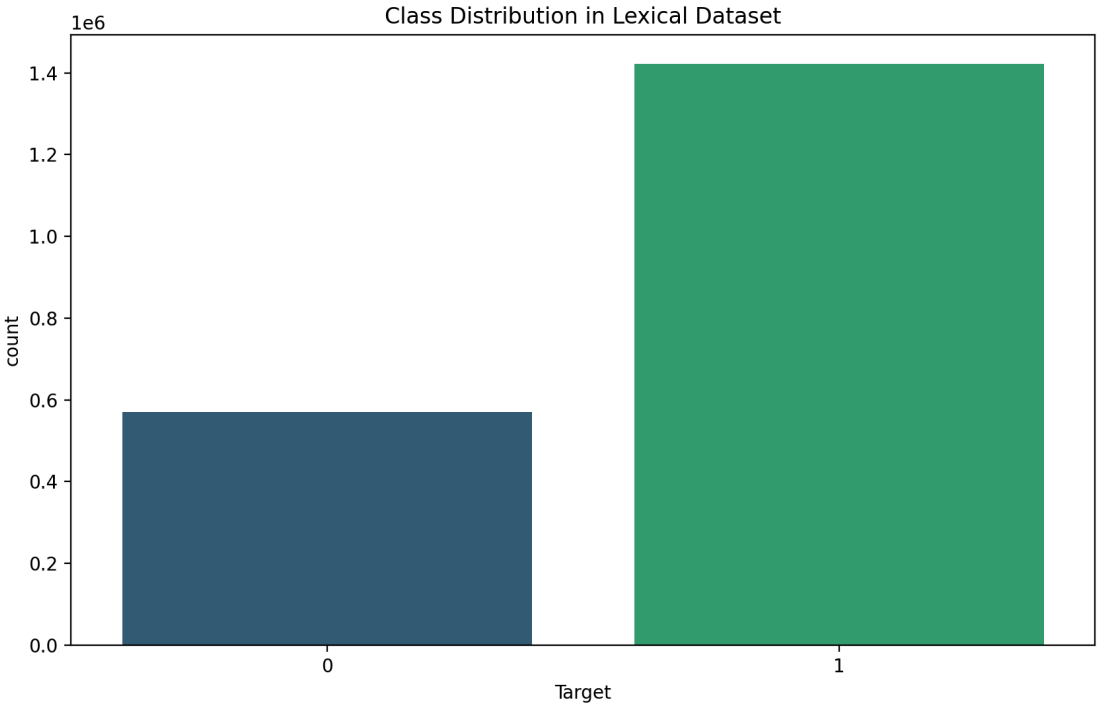


Figure 101: Lexical data class distribution

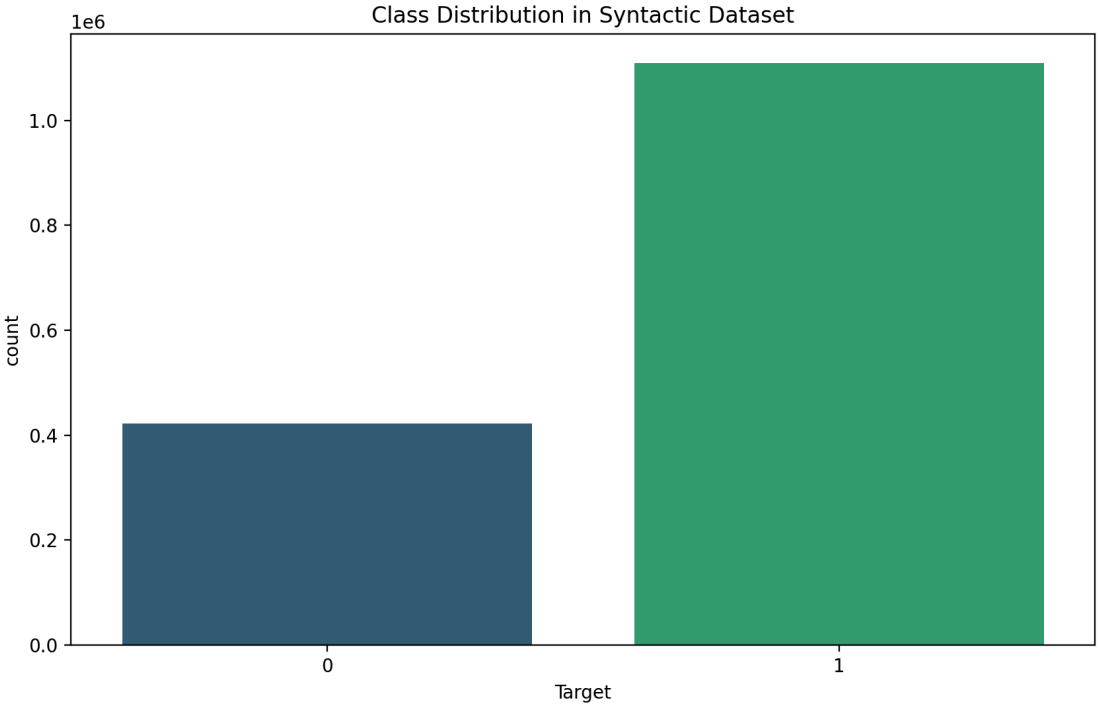


Figure 102:Syntactic data class distribution

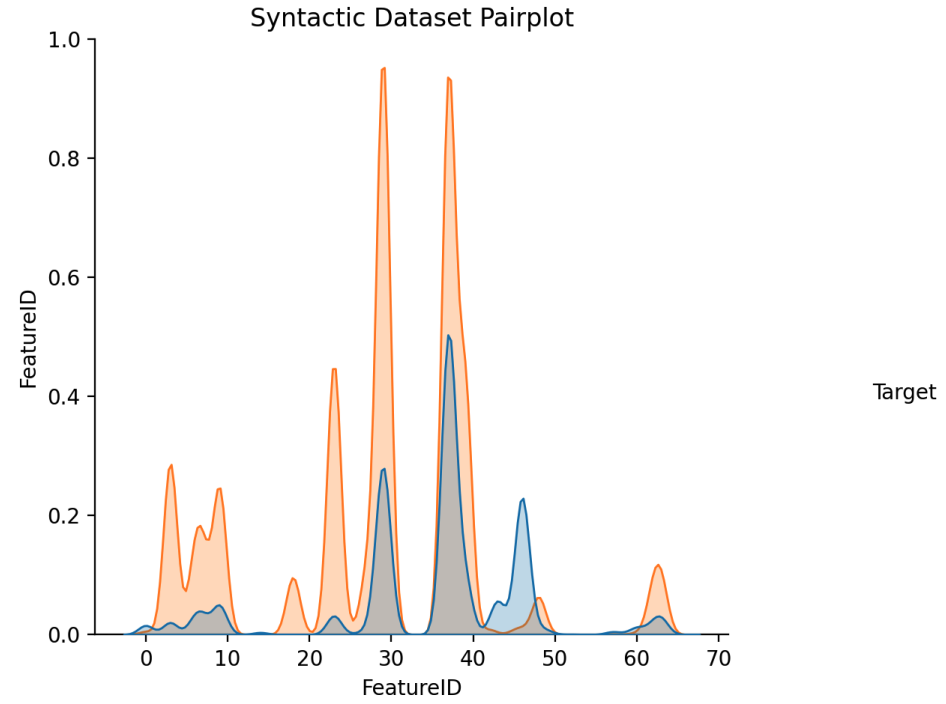


Figure 103: Syntactic dataset pairplot

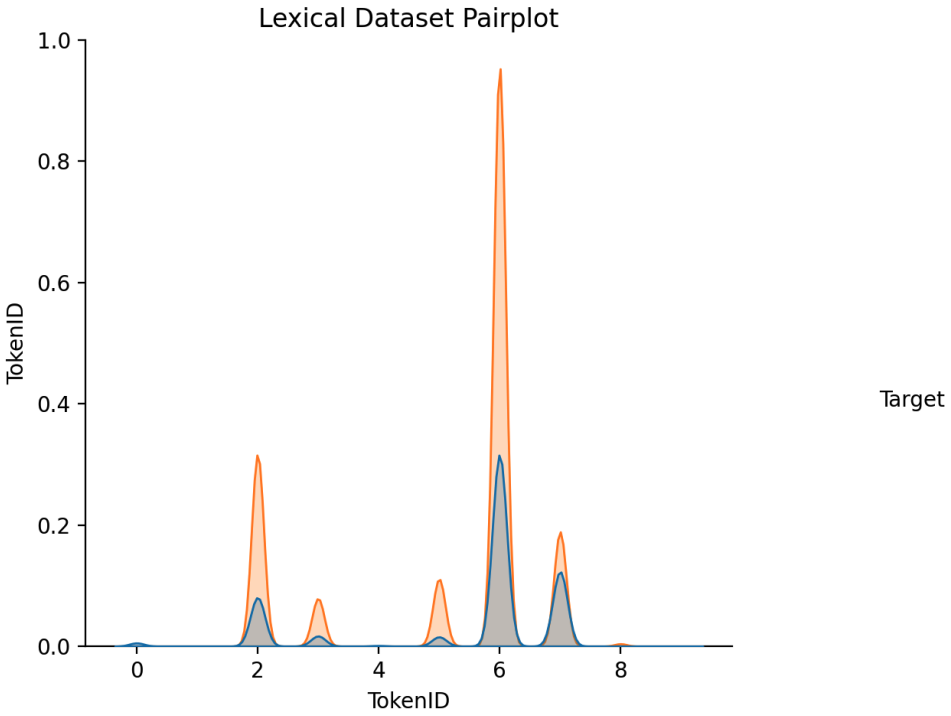


Figure 104: Lexical dataset pairplot

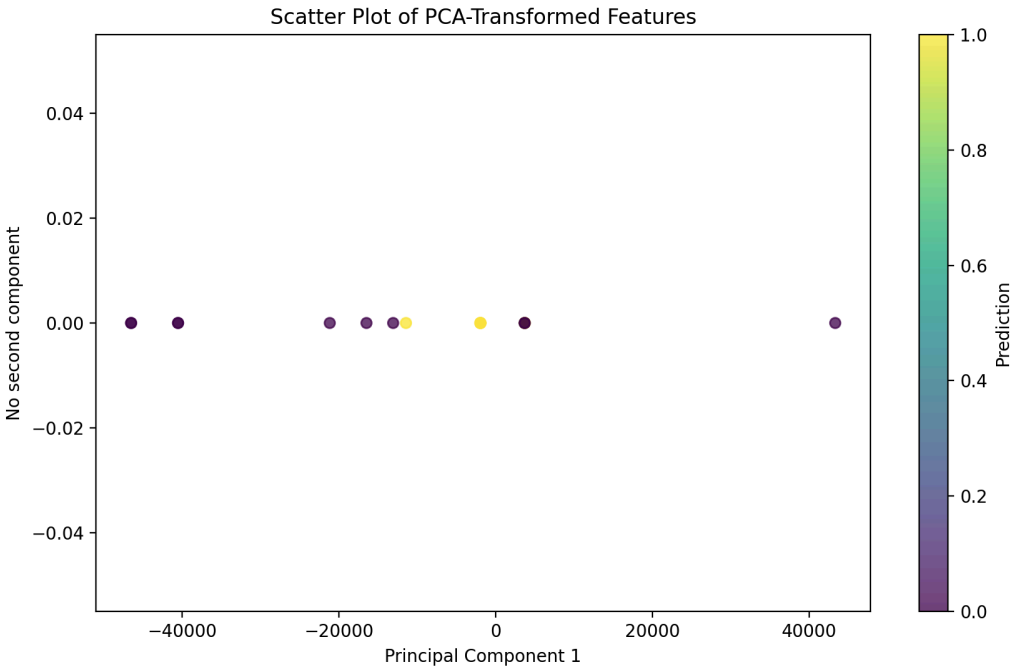


Figure 105: PCA transformed features scatter plot

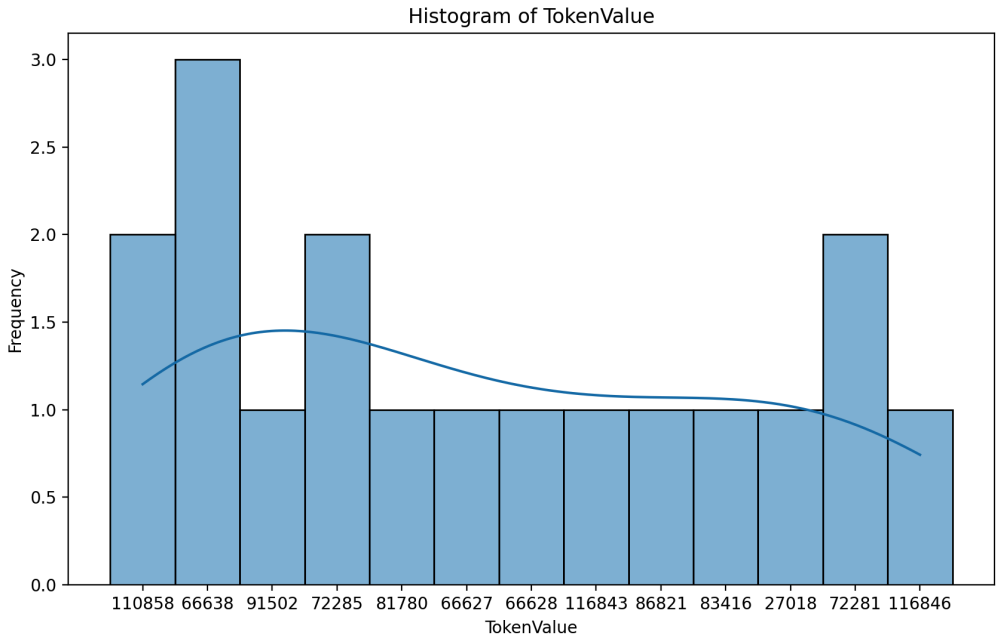


Figure 106: Tokenvalue Histogram

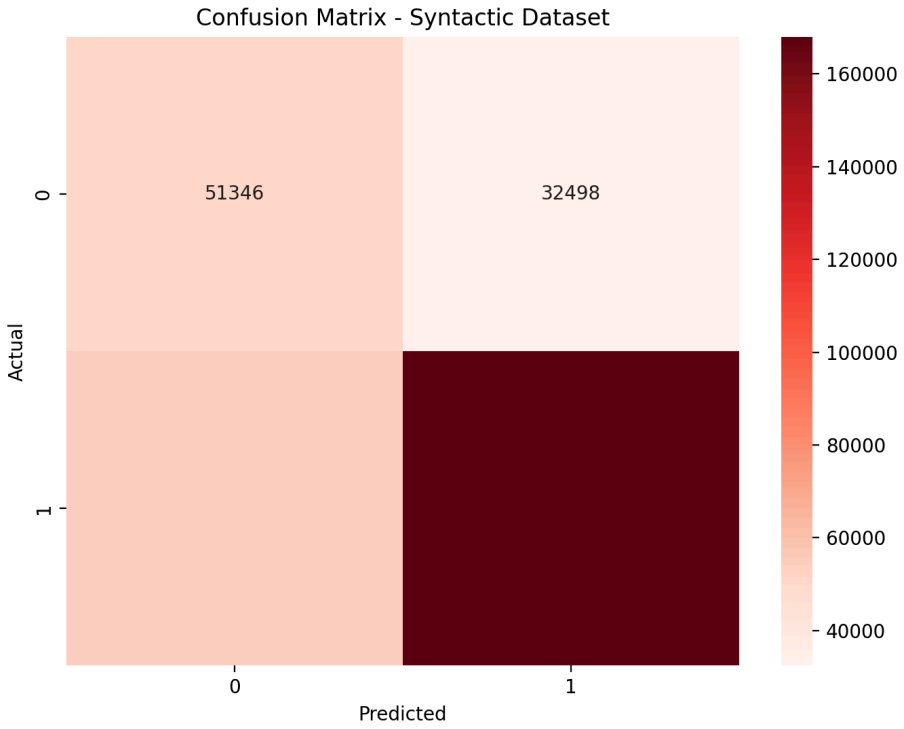


Figure 107: Syntactic dataset confusion matrix

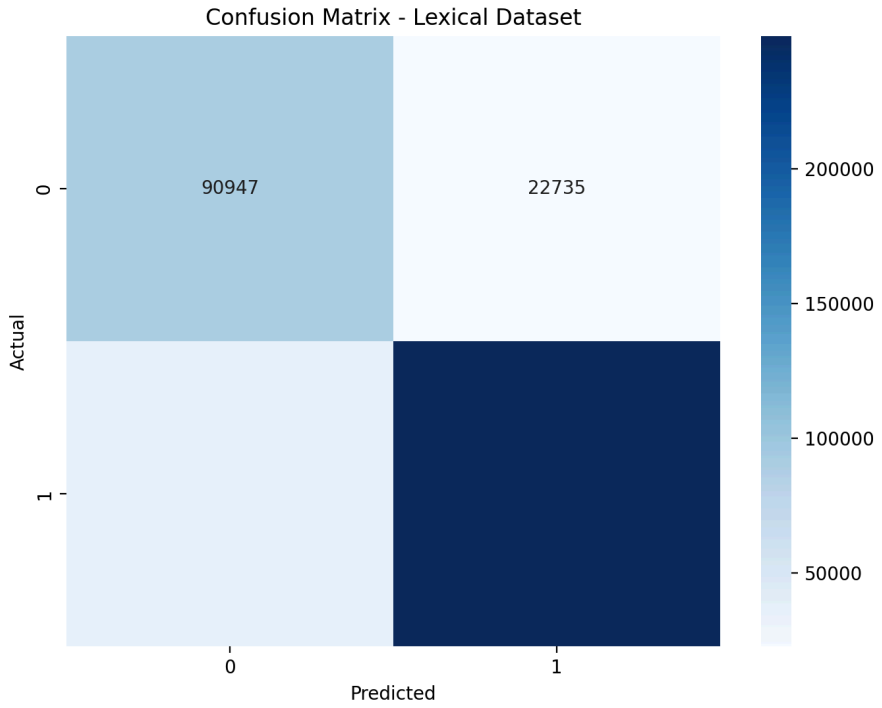


Figure 108: Lexical dataset confusion matrix

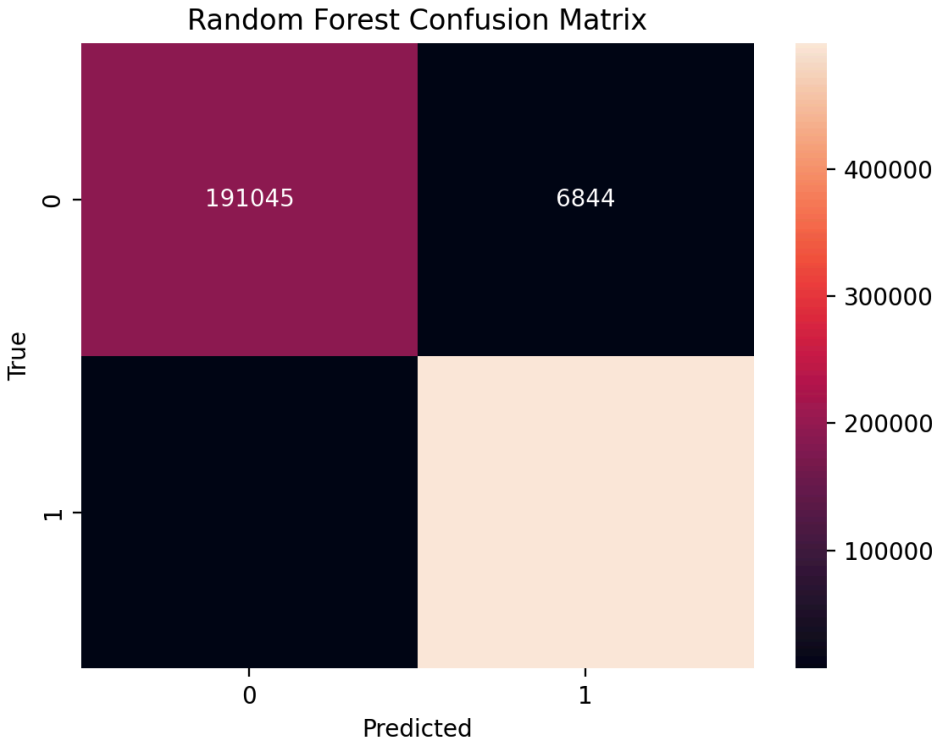


Figure 109: Random forest confusion matrix

Github:

https://github.com/ginnoro2/JS-MDA.git