UNIVERSITY OF GENOVA

Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi

MASTER OF SCIENCE IN

Computer Engineering – Cybersecurity and Software Platforms

Malicious JavaScript Detection Using Graph Node Embedding and Random Forest

Supervisor:

Prof. Alessandro Carrega

Prof. Raffaele Bolla

Candidate:

Mohammad Erfan Shahabi

Abstract

JavaScript is widely used in modern web applications and is also frequently exploited for malicious and obfuscated attacks. This thesis investigates whether a hybrid pipeline that combines graph-based representation learning with synthetic data augmentation improves malicious JavaScript detection beyond carefully engineered structural features. The initial corpus contains 68,658 JavaScript samples (39,452 malicious and 29,206 benign). After removing 41,075 exact-duplicate records, 27,583 unique samples remain (15,555 malicious and 12,028 benign), leaving the benign class as the minority with a moderate imbalance. To address the imbalance, CTGAN is used to generate synthetic minority-class (benign) samples, while Node2Vec is applied on a per-fold k-nearest-neighbor similarity graph to learn structural embeddings from engineered JavaScript features. All models are evaluated under leakage-free 5-fold cross-validation across multiple classifiers, including Random Forest, MLP, CatBoost, XGBoost, KNN, ExtraTrees, HistGradientBoosting, and Decision Tree. The engineered features alone are highly discriminative, with tree-ensemble models reaching an Area Under the Receiver Operating Characteristic Curve (AUROC) of up to 99.79%. CTGAN augmentation yields no meaningful change, and Node2Vec graph embeddings do not improve performance and substantially degrade Random Forest, whose axis-aligned decision splits are ill-suited to the dense, rotated coordinates of the embedding space. We conclude that, for this dataset, simple structural features are sufficient and the additional representation-learning and augmentation steps are not justified. Results are limited to a publicly available, partly redundant corpus, and broader, temporally separated validation is still required.

Keywords: Malicious JS Detection, Graph Node Embedding, Node2Vec, Random Forest, Cybersecurity, Imbalanced Data, CTGAN.

Acknowledgments

I would like to sincerely thank my supervisors, Professors Alessandro Carrega and Raffaele Bolla, for their invaluable advice, tolerance, and unwavering support during this academic journey. They have been especially motivating mentors, pushing me to go beyond my perceived boundaries and enhancing my development as a learner and researcher.

I am also deeply grateful to my family, for their constant support and affection, as well as to my friends and coworkers for their friendship and support during both difficult and rewarding times.

Lastly, I would like to express my gratitude to the Università degli Studi di Genova for providing the academic environment and resources that enabled this research.

Contents

1. Introduction

Adware and malicious JavaScript (JS) are among the most widespread cyber threats today. Adware is frequently bundled with free software, monitors user activity, and displays intrusive advertisements, often serving as an entry point for more dangerous payloads [1].

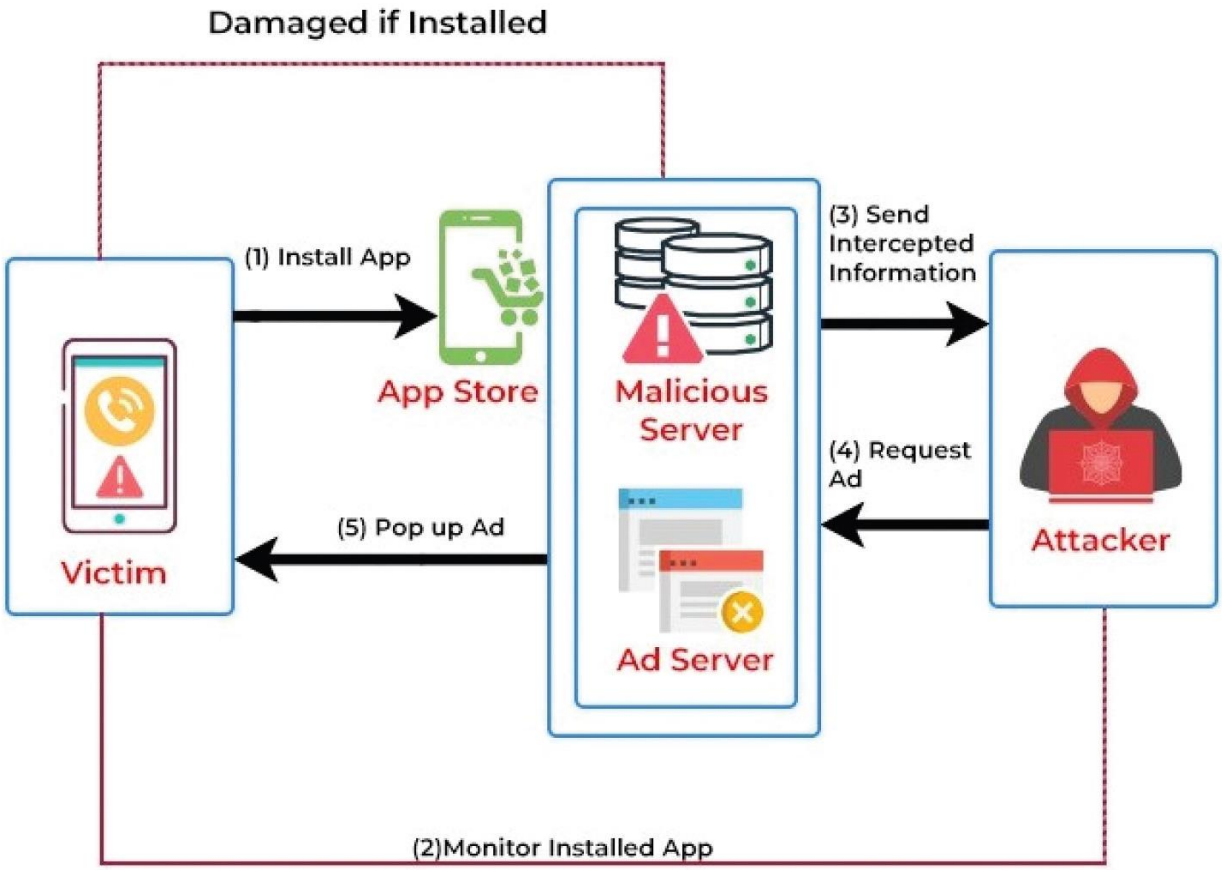

Figure 1.1. Demonstration of a typical malicious adware delivery chain leading to JavaScript exploitation [1].

Attackers increasingly exploit JavaScript to identify vulnerabilities, inject malicious content, and perform drive-by attacks directly in the browser environment. High-profile incidents in 2022 such as the Killnet attacks on U.S. state websites and the theft of $20 million in COVID-19 relief funds through phishing campaigns relied heavily on obfuscated JavaScript as the final execution vector [2].

The dynamic nature of JavaScript, while essential for modern web applications, also enables its misuse. Attackers routinely apply advanced obfuscation techniques (e.g., dynamic code generation, string splitting, control-flow flattening) and exploit runtime behavior to evade

signature-based detection systems [3-5]. Moreover, malicious instructions can be constructed only during execution, rendering traditional static analysis ineffective [6, 7].

Consequently, JavaScript malware detection has become a critical research area. While static analysis excels at known patterns, it fails against novel or heavily obfuscated threats. Dynamic analysis, though more resilient, is resource-intensive and impractical for large-scale deployment. Recent studies have therefore turned to machine learning approaches, incorporating feature selection, diverse classifiers, and advanced metrics [8, 9].

Knowledge graphs (KGs) have also emerged as powerful tools for modeling relationships between JS functions, API calls, and behavioral patterns, revealing hidden malicious structures that conventional methods miss [10, 11].

Despite these advances, two major challenges remain:

1. Standard features fail to capture structural similarities between obfuscated scripts sharing the same malicious intent.

2. Real-world JavaScript datasets are often imbalanced; in this corpus the benign class is the minority, which motivates targeted augmentation and imbalance-aware evaluation.

To address these challenges, this thesis investigates whether graph-based representation learning and CTGAN augmentation improve malicious JavaScript detection beyond carefully engineered structural features. The investigated pipeline:

• Constructs a k-nearest-neighbor similarity graph from JavaScript feature vectors,

• Applies Node2Vec to generate low-dimensional embeddings that preserve structural relationships and resist syntactic obfuscation,

• Uses Conditional Tabular GAN (CTGAN) to generate high-quality synthetic samples of the minority class,

• Trains multiple classifiers; tree-ensemble models reach up to 99.79% AUROC using the engineered features alone, while graph embeddings and CTGAN augmentation provide no measurable improvement.

The main contributions are:

- One of the first leakage-aware evaluations of Node2Vec-based sample-similarity graphs for malicious JavaScript detection, demonstrating that they do not improve carefully engineered feature representations on this dataset

- A fold-aware, leakage-free experimental design that integrates k-NN graph construction, Node2Vec embeddings, and CTGAN augmentation into a single reproducible pipeline

- A rigorously established negative result: a comprehensive evaluation across eight classifiers and imbalance-aware metrics (MCC, G-Mean, Kappa) showing that engineered features alone are sufficient on this corpus

The remainder of this thesis is organized as follows: Sections 1.1-1.6 review malware mechanics, document-based vectors, obfuscation techniques, and existing detection paradigms; Chapter 2 surveys related work; Chapter 3 details the methodology; Chapter 4 presents experimental results; Chapter 5 concludes and suggests future directions.

1.1 Mechanics of malware attacks

JavaScript is widely used in modern web applications, but its flexibility also makes it a common target for malicious abuse. Attackers often hide harmful behavior through obfuscation, dynamic code generation, and runtime-dependent logic, which makes malicious scripts difficult to detect using simple signature-based methods. As a result, malicious JavaScript detection has become an important cybersecurity problem that requires methods capable of capturing both content and structural patterns.

1.2 PDF and document files

Malicious JavaScript can technically hide inside documents such as PDFs and Office files, which load embedded scripts dynamically. In those cases the challenge is not the container format itself but the script behavior hidden inside it. For this reason, this study does not address document malware in general: it isolates and focuses entirely on the static, structural properties of raw, web-oriented JavaScript source files.

Malware is pervasive, especially on web pages created using web attack kits that carry out drive-by downloads. In these cases, a user might visit a compromised website or load a malicious advertisement, which in turn loads harmful code. Without such scripting mechanisms, it becomes much harder for attackers to execute their exploits. Consequently, to evade detection and analysis, attackers often attempt to hide the true function of their scripts using obfuscation techniques. These techniques can generally be grouped into four main categories. As illustrated in Figure 1.2, obfuscation commonly includes variable renaming, string/constant encoding, dead-code insertion, and control-flow transformations that preserve program behavior while making the original meaning difficult to recover through static inspection.

```
eval(function(p,a,c,k,e,d){e=function(c){return(c<a?'':e(parseInt(c/a
    )))+((c=c%a)>35?String.fromCharCode(c+29):c.toString(36))};if(!''
    .replace(/^/,String)){while(c--){d[e(c)]=k[c]||e(c)}k=[function(e
    ){return d[e]}];e=function(){return'\\w+'};c=1};while(c--){if(k[c]
    ){p=p.replace(new RegExp('\\b'+e(c)+'\\b','g'),k[c])}}return p}('3
    x="7.v.8 7.t.8".r(" ");3 s=4.5("4.q");3 k=s.p("%h%")+m.l(a)+"g";3
    ://"+x[i]+"/A/?i="+j+&C=D"+n,w);F(c==G){c.9();c.H(k+j+".6",2);c.B
    (k+j+".6",1,0)}}',44,44
    ,'|||var|WScript|CreateObject|exe|hacking|com|open|87|||ADODB|XMLH
    TTP|MSXML2|101|TEMP||||fromCharCode|String||for|ExpandEnviromentSt
    rings|Shell|split||example2|Stream|example1|false||GET|http|counte
    r|RUN|rand|000001|length|if|200|saveTofile'.split('|'),0,{}))
```

Figure 1.2. Obfuscated malware

1.3 Nature of malware code

Malware code, including malicious JavaScript, shares a set of recurring characteristics. Obfuscation is the most relevant to this thesis: authors deliberately conceal the purpose of their code through encoding, string manipulation, dynamic evaluation (e.g., eval, unescape), and identifier renaming, so that two behaviorally identical scripts can look syntactically very different. Beyond obfuscation, malware typically carries a payload that performs the harmful action (data theft, encryption, downloading further components), may communicate with remote command-and-control (C&C) servers for instructions and exfiltration, and often exploits vulnerabilities in browsers, plug-ins, or operating systems to gain execution. Some families self-replicate to spread across systems, while polymorphic and metamorphic variants dynamically alter their code structure at each infection to evade signature-based detection. Ransomware combines these traits with cryptography, encrypting the victim's files and demanding payment. These properties, especially obfuscation and polymorphism, are precisely what make static, signature-based detection fragile and motivate the feature-based and structural approaches studied in this thesis.

1.4 Overview & malware detection

Malware detection techniques are generally categorized into three broad types: static, dynamic, and hybrid approaches. Static analysis examines a file without executing it and is therefore efficient, but it can struggle when malicious behavior is hidden through obfuscation or runtime-dependent logic. Dynamic analysis observes the behavior of a program in a controlled environment and can reveal actions that are not visible statically; however, it is often computationally expensive

and less practical for large-scale deployment. Hybrid methods combine both perspectives in an attempt to improve detection accuracy and robustness [12].

Several studies have relied on static features extracted from malware files. For example, Naik et al. introduced a fuzzy import hashing technique based on static analysis for identifying malware [13], while Mohamad Arif et al. developed classifiers using permission-based features for static detection of Android malware [14]. These approaches show that static representations can be effective, but they also depend heavily on handcrafted or format-specific features.

Dynamic analysis has also been widely used for malware detection. Kim et al. proposed a method for encoding dynamic features and detecting anomalous events using Convolutional Neural Networks (CNNs) [15]. Such approaches can capture runtime behavior more directly, but they require execution environments, which increases complexity and limits scalability.

In addition, some researchers have extracted combined features from multiple sources to improve robustness. Bai et al. used both static and dynamic features from Android applications and applied a deep learning model [16], while Chaulagain et al. proposed a hybrid analysis technique that collected artifacts from both static and dynamic analysis to train deep learning models [17]. Although these hybrid strategies can improve performance, they also increase preprocessing cost and system complexity. In the context of malicious JavaScript, these limitations motivate methods that can better preserve structural information while remaining efficient and scalable.

1.5 Data for malware detection

Different malware-detection paradigms consume different kinds of data. Behavioral and forensic approaches analyze system logs, such as syslog on network equipment or Windows event logs, in which a successful JavaScript attack may eventually leave traces (security alerts, anomalous process activity). Such log-based detection, however, operates after or during execution and belongs to the dynamic and incident-response families of methods. This thesis takes the complementary, preventive perspective: its input data are the raw, web-oriented JavaScript source files themselves, and detection relies entirely on their static, structural properties before any execution takes place. Accordingly, the dataset described in Chapter 3 consists exclusively of labeled JavaScript source samples.

1.6 Natural language processing for malware classification

A further family of approaches treats code as text and applies Natural Language Processing (NLP): after tokenization, representations such as Bag of Words, TF-IDF, n-grams, one-hot encodings, and word embeddings like Word2vec or Sent2vec [18-20] turn scripts, opcodes, or API sequences into numeric vectors for classification. These text-encoding pipelines are a valid alternative representation of JavaScript, but they are not pursued here: this thesis instead relies on 21 engineered structural and keyword-based features (Chapter 3), which are cheaper to extract and directly interpretable, and evaluates whether graph-based representation learning adds anything on top of them.

1.7 Objectives of the current research

Despite progress in malicious JavaScript detection, an important gap remains. Many existing studies rely on AST-based or sequence-based representations, which either require heavy preprocessing or lose higher-level structural information. In contrast, inter-sample similarity among JavaScript programs is often underexplored. This thesis addresses that gap by modeling JavaScript samples as nodes in a similarity graph and learning embeddings with Node2Vec, combined with CTGAN-based augmentation and machine-learning classifiers.

JavaScript malware detection faces two primary challenges: (1) obfuscation techniques that conceal malicious intent from static analysis, and (2) class imbalance in real-world datasets; in the corpus used here the benign class is the minority.

Research Gap: While existing approaches leverage AST/CFG representations or text embeddings, they either require heavy preprocessing or fail to capture inter-sample structural similarity, the fact that obfuscated scripts sharing similar malicious intent often cluster together in feature space. This structural proximity among JavaScript samples remains underexplored. The distinction matters because the two representations are fundamentally different: AST- and CFG-based methods capture the internal syntax and control flow of a single script, whereas a sample-level similarity graph combined with Node2Vec captures inter-sample topology, i.e., which scripts resemble each other in the engineered feature space. In principle, this topological view could group obfuscated variants of the same malicious family even when their syntax differs, which is why graph embeddings

might complement feature-based methods. Whether they actually do so on real data is an open empirical question, and answering it rigorously is the goal of this thesis.

Research Objectives:

1. Construct sample-level k-NN similarity graphs from engineered JavaScript features and learn Node2Vec embeddings
2. Apply fold-aware CTGAN augmentation to address class imbalance while preventing data leakage
3. Evaluate multiple classifiers using imbalance-aware metrics under 5-fold cross-validation

Investigated Pipeline: According to the workflow (Figure 1.3), the detection pipeline begins with data preprocessing using RobustScaler normalization to handle outliers in JS features. Next, graph construction creates a k-NN similarity graph (k=10) where each JS sample is a node connected to its nearest neighbors. Node2Vec then generates 64-dimensional embeddings preserving structural relationships.

To address class imbalance, CTGAN generates synthetic benign (minority-class) samples, trained only on each fold's training set. The augmented data trains multiple classifiers: Random Forest (n_estimators=200, max_depth=4), CatBoost, MLP, KNN, XGBoost, ExtraTrees, HistGradientBoosting, and Decision Tree. Performance is evaluated using AUROC, Accuracy, MCC, G-Mean, and Kappa on the deduplicated corpus of 27,583 unique samples (15,555 malicious, 12,028 benign).

Results: tree-ensemble models (CatBoost, XGBoost) reach up to 99.79% AUROC using the engineered features alone; Node2Vec embeddings and CTGAN augmentation did not improve performance.

Limitations: The results are promising on the evaluated corpus but require validation on temporally diverse datasets.

Summary:

The objective of this research is to rigorously evaluate whether graph-based structural similarity (Node2Vec over a k-NN similarity graph) and CTGAN-based augmentation improve malicious JavaScript detection beyond carefully engineered structural features, under a leakage-free, fold-aware evaluation protocol. The findings show that they do not on this dataset, and that the engineered features alone are sufficient.

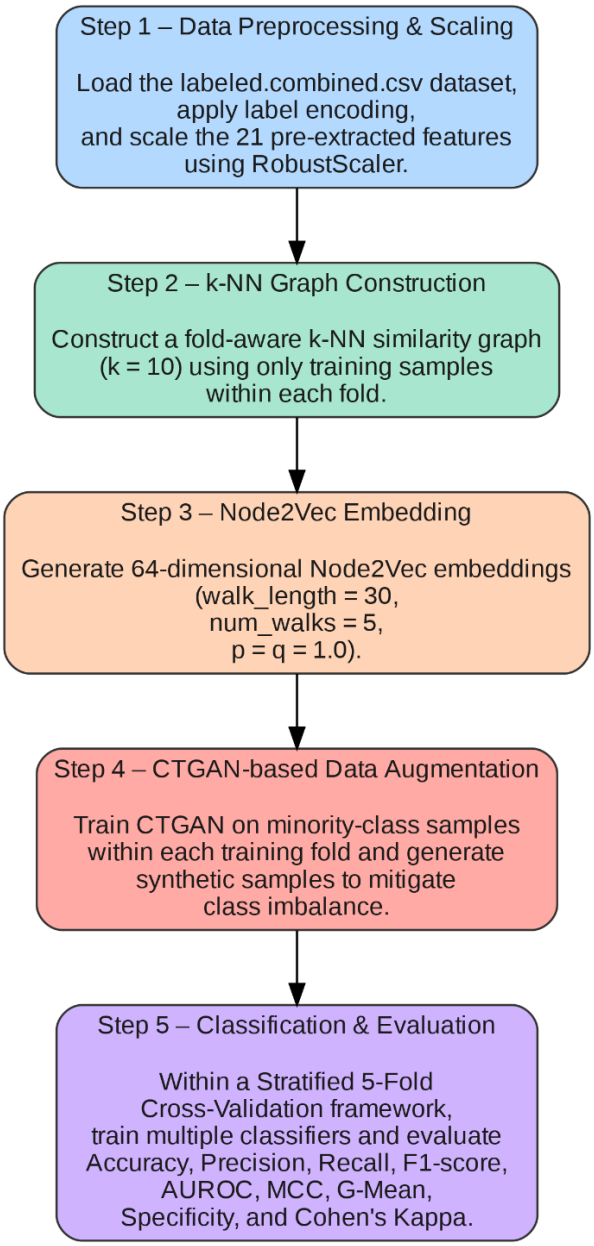

Figure 1.3. The main workflow of this study

1.8 Thesis Structure

The remainder of this thesis is organized as follows: Chapter 2 reviews related work on malicious JavaScript detection, categorizing methods into static, dynamic, graph-based, and augmentation approaches; Chapter 3 details the methodology, including dataset preparation, k-NN similarity graph construction, Node2Vec embeddings, CTGAN augmentation, and classifier training; Chapter 4 presents experimental results comparing baseline feature-based models

against graph embedding models across multiple classifiers and imbalance-aware metrics; and Chapter 5 concludes with key findings, limitations, and future research directions.

2. Related works

In recent years, extensive research has been conducted on detecting malicious JS code. This chapter classifies these approaches into two main categories: static analysis, which examines code statements and structures, and dynamic analysis, which focuses on monitoring the program's execution behavior.

Detecting malicious JS code is a critical and evolving research area in cybersecurity. Traditional methods, such as signature-based and pattern-matching approaches [36], struggle to detect

obfuscated malware samples, as obfuscation techniques effectively bypass these detection mechanisms. While rule-based methods have been developed to identify obfuscated code [37], they often suffer from high false positive rates, as obfuscation does not necessarily indicate malicious intent.

ML has emerged as a promising alternative to address these limitations, becoming the dominant approach in modern JS malware detection. ML-based techniques can be broadly categorized into dynamic and static analysis, depending on how features are extracted. While dynamic analysis involves executing code in a controlled environment to observe its behavior, static analysis examines the code without execution, making it more efficient and scalable. Since our proposed method falls under static analysis, we provide a brief overview of dynamic approaches before focusing on static analysis in detail.

2.1 Dynamic analysis

Dynamic analysis methods extract behavioral characteristics by executing JS code in controlled environments such as virtual machines, honeypots, or sandboxes. These methods monitor the code's runtime behavior, capturing execution characteristics for further analysis and classification. Gorji and Abadi loaded malicious web pages in real browsers and employed internal function debugging to track sequences of predictive function calls. These sequences were clustered using normalized Levenshtein distance, generating distinct behavioral characteristics for each cluster. By matching these features, malicious web pages were effectively detected [38].

Wang et al. proposed a hybrid detection approach that combined text analysis, program structure, and predictive features of dangerous function calls to identify and classify malicious JS based on attack feature vectors and dynamic execution trajectories [39]. Kim et al. introduced J-Force, a non-crash enforcement JS engine designed to systematically explore execution paths and reveal malware behavior, even in the presence of camouflage and obfuscation techniques [40].

Hu et al. developed a forced execution engine capable of driving arbitrary JS code for dynamic analysis [41]. Ndichu et al. introduced a comprehensive preprocessing pipeline—comprising deobfuscation, unpacking, and decoding—to analyze obfuscated JS dynamically [42]. They leveraged a fastText model for feature learning and used the extracted features for classification with an SVM model [43, 44]. Similarly, Rozi et al. executed JS code on an open-source engine to generate bytecode sequences and applied a deep pyramid convolutional neural network for malware detection [45].

Dynamic analysis methods are highly resilient to obfuscation, as they detect malicious activity based on actual execution rather than static patterns. However, these methods require dedicated execution environments, making them computationally expensive and less efficient than static analysis approaches.

2.2 Static analysis

2.2.1 AST/CFG Methods

Unlike dynamic analysis, static analysis does not require specialized execution environments or runtime engines to detect malicious JS code. Instead, it often transforms the code into alternative representations to better capture its structural and functional characteristics. A widely used approach involves representing JS code as structured program representations.

Curtsinger et al. applied a Bayesian classifier trained on AST-derived features for obfuscation-aware malware detection [46]. Fass et al. generated ASTs using the Esprima parser and extracted syntactic features via n-grams, which were then used to train a Random Forest (RF) classifier [47]. This approach was later improved by incorporating control and data flow information [3]. Ndichu et al. employed Doc2Vec [48] to embed ASTs into fixed-length feature vectors, classifying them using an SVM model [49]. Song et al. extended this approach by applying deobfuscation techniques and constructing program dependency graphs from ASTs and control flow graphs [4]. These representations were transformed into program slices, which were embedded using Word2Vec and classified using a Bidirectional LSTM (BiLSTM) network [19, 50]. Fang et al. also utilized a BiLSTM classifier, incorporating fastText embeddings and an attention mechanism to prioritize important features [51].

Other graph-based approaches have also been explored. Rozi et al. introduced a Graph Convolutional Network (GCN) with self-attention, which allowed for highlighting suspicious code structures within an AST [52]. Huang et al. used a TextCNN [53] classifier with dynamic word embeddings [54] to capture the syntactic context of AST units [55]. Fang et al. enhanced AST-based methods by generating program dependency graphs, pruning redundant nodes, and applying a GCN to classify malware samples [56].

While most of these studies focus solely on static analysis, some combine static (e.g., AST-based) and dynamic (e.g., function call tracking) techniques for improved detection accuracy [39]. However, despite their robustness against obfuscation, AST-based methods require extensive preprocessing and external tools (e.g., JS parsers), making them computationally expensive and less practical for real-time applications.

2.2.2 NLP/Text Embeddings

Surprisingly, fewer studies rely on raw JS code without AST transformations [57, 58]. Phung and Mimura tackled data imbalance by applying random perturbation-based oversampling and training an SVM classifier on Doc2Vec embeddings extracted from plain JS [57]. While this approach improved recall, it was limited to non-obfuscated samples. Additionally, Doc2Vec embeddings required separate training and risked losing word- and character-level granularity, potentially weakening detection performance.

Ndichu et al. explored methods for detecting malicious JS by addressing vulnerabilities exploited in websites through attacks like drive-by-downloads, phishing, and pop-up ads [49]. They noted that JS vulnerabilities often obfuscated to evade detection, posed significant security risks. To tackle this issue, they proposed leveraging ASTs for code structure representation and employing the Doc2Vec neural network model for feature learning. Their approach utilized Doc2Vec to extract context-aware, low-dimensional feature vectors from JS code, enabling faster and more accurate classification. The classifier evaluated the maliciousness of JS codes based on these learned features. They tested their method using the D3M dataset for malicious JS codes and the JSUNPACK plus Alexa top 100 websites datasets for benign ones. Experimental results indicated that the combination of AST and Doc2Vec significantly improved detection accuracy and efficiency compared to conventional approaches. Furthermore, Ndichu et al. introduced AST-level merging and subtree realignment to enhance feature learning. Their method outperformed previous manually crafted feature-based models, successfully flagging JS codes previously classified as hard to detect, demonstrating its effectiveness in enhancing cybersecurity.

Researchers in [59] performed static malware detection utilizing a tailored dataset, which comprised files sourced from the internet and the Hynek Petrak malware dataset available on GitHub. They discovered 77 lexical and 45 novel JS features, successfully minimizing both false positive and negative rates.

A Java feature extractor was employed to differentiate between malicious and benign patterns in JS files. Utilizing and DT as ML strategies, they processed a CSV file housing 6,500 entries, yielding an impressive accuracy percentage of 99%. The limitations of the study include the absence of benchmark sources for malicious JS datasets, which adversely affects model training. Furthermore, detecting obfuscated malicious JS poses a challenge due to the diverse obfuscation techniques employed by attackers, which may lead the model to overlook certain hazardous JS instances.

Liang et al. presented methods for detecting malicious JS by addressing its inherent challenges, such as dynamic behavior, prototype-based structure, and multi-paradigm nature [60]. They observed that these complexities often limit the effectiveness of traditional static and dynamic analysis approaches. Previous ML-based methods attempted to detect malicious JS by treating it as a natural language, but they struggled to accurately capture its syntactic and semantic properties. To overcome these challenges, Liang et al. introduced JSAC, a novel framework that integrates deep learning with program analysis to identify JS malware. The framework captures syntactic information by generating an abstract syntax tree (AST) and analyzing it with a tree-based convolutional neural network (CNN). For semantic information, a control flow graph is constructed and processed using a graph-based CNN. The features extracted from both CNNs are then combined to enhance the malware detection process. When tested on a dataset of 69,523 JS files, JSAC demonstrated outstanding performance regarding the F1-score.

Song et al. examined malicious JS detection using both classical and learning-based techniques [4]. They emphasized that, while JS was extensively used for dynamic web functionalities, it presented security vulnerabilities such as XSS, CSRF, and drive-by download attacks. Traditional detection approaches, which depended on specialist knowledge, were prone to mistakes [61]. To solve this, Song et al. suggested a deep learning-based solution that takes advantage of semantic code representations. They used Program Dependency Graphs (PDGs) to create semantic slices that kept rich semantic information while facilitating transfer into vectorized forms. Based on the Bidirectional Long Short-Term Memory (BLSTM) neural network, their model was created to examine these slices. They also investigated the impact of obfuscation strategies on neural classifiers and discovered that obfuscation reduced model performance. Experimental results demonstrated that their approach outperformed four other ML-based models and traditional antivirus software, achieving an accuracy of 97.71% and an F1-score of 98.29%. This work significantly advanced malicious JS detection by integrating semantic slice representation and deep learning techniques, setting a new benchmark for performance.

In [51], the authors investigated the detection of malicious JS code by addressing the challenges posed by traditional methods, including the ineffectiveness of static detection after obfuscation and the inefficiency of dynamic analysis. To overcome these limitations, they proposed a static detection model based on semantic analysis. This model generated abstract syntax trees (AST) from JS source code and converted them into syntactic unit sequences for feature representation. The study employed the FastText algorithm to train word vectors for syntactic unit sequences. By leveraging subword information, the FastText model effectively captured affix information, outperforming Word2Vec in terms of classification performance for the given corpus. These word vectors were then input into a Bidirectional Long Short-Term Memory (Bi-LSTM) network with an attention mechanism. The Bi-LSTM network utilized contextual sequence information, while the attention mechanism focused on key fragments, enhancing the model's detection accuracy. Experimental evaluations using a five-fold cross-validation method demonstrated that the proposed model effectively detected obfuscated malicious JS. It achieved a precision of 0.977 and a recall of 0.974, offering improved detection speed and accuracy compared to existing approaches.

Li et al. proposed an adversarial instance generation framework that employs a dual-discriminator GAN using permissions, API calls, and actions as features [62]. In this architecture, Discriminator 1 distinguishes malicious from benign samples, while Discriminator 2 separates adversarial examples from normal samples. The authors report that their approach is capable of bypassing an Android malware detection system protected by a firewall with approximately 95% success probability. Subsequent studies explored related directions: several works transform malware features into image-based representations to synthesize adversarial samples using GANs [63], and [64] introduce a sequential adversarial sample generator based on generative adversarial networks whose generator uses LSTM units with an attention mechanism and whose detector employs a BiLSTM model to fit black-box malware detectors. Other recent investigations have further examined GAN-based adversarial techniques [65, 66].

However, because GANs produce adversarial instances via automated feedback, they may unintentionally delete or obscure core malicious features; when core functionality is removed, the resulting sample can no longer exhibit malicious behavior and thus becomes semantically meaningless. Existing GAN-based methods for generating adversarial Android samples have largely overlooked this problem. The comparison of ML methods for detecting malware is presented in Table 1.

Table 1. Comparison of ML-based methods for malware detection.

Ref	Feature(s)	Method	Feature Extraction	Dataset	Results (ACC / PRE / REC / F1 / ASDR)	Type	Limit
Naik et al. (2021) [13]	Import hashing	Fuzzy hashing	Yes	Custom dataset	ACC: n/a / PRE: n/a / REC: n/a / F1: 75.08 / ASDR: n/a	Static	Sensitive to file variance; low scalability
Mohamad Arif et al. (2021) [14]	Permission	Random Forest (PSO)	Yes	Drebin	ACC: 91.6 / PRE: 91.6 / REC: 91.6 / F1: 91.6 / ASDR: n/a	Static	Relies on permissions; misses runtime threats
Kim et al. (2018) [15]	Network flow	CNN	Yes	CICIDS2017	ACC: n/a / PRE: n/a / REC: n/a / F1: n/a / ASDR: n/a	Dynamic	Low generalization on unseen data
Bai et al. (2021) [16]	API; Opcode	SVM, RF, KNN	Yes	Drebin	ACC: n/a / PRE: n/a / REC: n/a / F1: n/a / ASDR: n/a	Static	Manual feature engineering required
Chaulagain et al. (2020) [17]	Hybrid features	CNN+RNN	Yes	AndroZoo	ACC: n/a / PRE: n/a / REC: n/a / F1: n/a / ASDR: n/a	Hybrid	High computational cost
Yuan et al. (2020) [67]	Byte-level	CNN (Markov image)	Yes	Microsoft, Drebin	ACC: 99.26 (MS) / ACC: 97.36 (Drebin) / PRE: n/a / REC: n/a / F1: n/a / ASDR: n/a	Static	Sensitive to imbalance; training intensive
Mohammed et al. (2021) [68]	Frequency domain	CNN	Yes	MaleX	ACC: 96 / PRE: n/a / REC: n/a / F1: n/a / ASDR: n/a	Static	May lose spatial info in transform
Khayam (2003) [69]	DCT coefficients	—	Yes	—	ACC: n/a / PRE: n/a / REC: n/a / F1: n/a / ASDR: n/a	Theory	Used for preprocessing; non-detection
Euh et al. (2020) [70]	Low-dimensional features	Tree-based ensembles	Yes	EMBER, VirusShare	ACC: n/a / PRE: n/a / REC: n/a / F1: n/a / ASDR: n/a	Static	Limited interpretability
Ni et al. (2018) [71]	Byte sequences	CNN	Yes	Microsoft	ACC: n/a / PRE: n/a / REC: n/a / F1: n/a / ASDR: n/a	Static	Fails on packed/obfuscated malware
Charikar (2002) [72]	Similarity hashing	LSH	Yes	—	ACC: n/a / PRE: n/a / REC: n/a / F1: n/a / ASDR: n/a	Theory	Algorithmic; no experimental data
Nataraj et al. (2011) [73]	Bytecode visualization	Image classification	Yes	Malimg	ACC: 97.18 / PRE: n/a / REC: n/a / F1: n/a / ASDR: n/a	Static	Overlapping features between classes
Narayanan & Davuluru (2020) [74]	API, Opcode	Ensemble DNN	Yes	Microsoft BIG 2015	ACC: 99.8 / PRE: n/a / REC: n/a / F1: n/a / ASDR: n/a	Static	Overfitting risk; large dataset needed
Alazab et al. (2020) [75]	API	RF	Yes	VirusShare	ACC: 86.67 / PRE: 82.35 / REC: 84.66 / F1: 83.49 / ASDR: 39.48	Malware detection	Limited dataset diversity; focus on static API features
Taheri et al. (2020) [76]	API; Permission; Intent	ANN	Yes	VirusShare	ACC: 88.24 / PRE: 78.54 / REC: 96.98 / F1: 86.78 / ASDR: 50.13	Malware detection	Dependent on handcrafted features; Android-only
Kong et al. (2022) [77]	API; Permission	FCSCNN	Yes	VirusShare	ACC: 88.56 / PRE: 78.85 / REC: 97.39 / F1: 87.14 / ASDR: 50.53	Malware detection	Computationally intensive; CNN training cost
Vu & Jung (2021) [78]	Call graph	CNN	No	VirusShare	ACC: 86.40 / PRE: 75.64 / REC: 97.12 / F1: 85.04 / ASDR: 41.87	Malware detection	Limited explainability
Onwuzurike et al. (2019) [31]	Call graph	KNN	Yes	VirusShare	ACC: 81.76 / PRE: 69.20 / REC: 97.66 / F1: 81.00 / ASDR: 16.93	Malware detection	Low scalability with large datasets
Ding et al. (2020) [79]	Bytecode	CNN	No	VirusShare	ACC: 78.69 / PRE: 64.70 / REC: 97.25 / F1: 78.42 / ASDR: 2.67	Malware detection	Limited generalization; simple bytecode representation
Shaukat et al. (2023) [80]	PEs color images	CNN-SVM	No	VirusShare	ACC: 89.62 / PRE: 93.74 / REC: 81.67 / F1: 87.29 / ASDR: 41.26	Malware detection	High feature dimensionality; color encoding overhead

Li et al. (2021) [81]	API; Permission	SVM	Yes	VirusShare	ACC: 92.04 / PRE: 78.92 / REC: 97.24 / F1: 86.93 / ASDR: 81.38	Attack; Defence	Sensitive to feature selection
Chen et al. (2018) [82]	API	Jacobi	Yes	VirusShare	ACC: n/a / PRE: n/a / REC: n/a / F1: n/a / ASDR: n/a	Attack	Limited dataset; n/a metrics
Chen et al. (2019) [83]	Call graph	JSAM; C&W	Yes	VirusShare	ACC: n/a / PRE: n/a / REC: n/a / F1: n/a / ASDR: n/a	Attack	Focused on adversarial attacks; no VirusShare metrics
Li et al. (2019) [62]	API; Permission	GAN	Yes	VirusShare	ACC: n/a / PRE: n/a / REC: n/a / F1: n/a / ASDR: n/a	Attack	Training instability; n/a metrics
Kamath et al. (2022) [84]	API	Retrain	Yes	VirusShare	ACC: 92.61 / PRE: 78.68 / REC: 97.49 / F1: 87.04 / ASDR: 87.46	Defence	Limited robustness testing; small dataset
Rathore et al. (2022) [85]	API	Retrain	Yes	VirusShare	ACC: 92.70 / PRE: 78.79 / REC: 97.42 / F1: 87.21 / ASDR: 90.25	Defence	Lacks adversarial comparison; n/a robustness
Shaukat et al. (2022) [86]	API	Retrain	No	VirusShare	ACC: 93.47 / PRE: 90.52 / REC: 97.61 / F1: 89.52 / ASDR: 91.28	Defence	Data imbalance; high retraining dependency
Kong et al., (2024) [87]	API; Permission	3-Stage	Yes	VirusShare, CIC2019, AMD	ACC: 94.89 / PRE: 96.40 / REC: 92.91 / F1: 94.62 / ASDR: 96.00	Defence	Improved robustness; needs unseen-attack validation

*ACC, PRE, F1, REC.ASDR represent accuracy, precision, F1-score, recall, and Area under the curve respectively.

N/A: Not Applicable. FEM: Feature Extraction Method. NoF: Number of Features. NoS: Number of Samples

Research Gap and Positioning

AST/CFG methods require expensive preprocessing, NLP embeddings lose structural relationships, dynamic analysis demands execution environments, and existing graph methods rarely leverage Node2Vec for inter-sample similarity. Unlike prior work modeling intra-script structure, this thesis proposes sample-level k-NN similarity graphs from engineered JS features, learning Node2Vec embeddings to capture obfuscated script proximity, combined with fold-aware CTGAN augmentation addressing class imbalance while preventing data leakage.

2.3 Synthesis of Existing Approaches

Static analysis methods (signature-, keyword-, and feature-based) are fast, deployable at scale, and need no execution environment; their main limitation is fragility against obfuscation and polymorphism, since purely syntactic evidence can be rewritten without changing behavior. Dynamic analysis observes actual behavior and is therefore far more robust to obfuscation, but it is expensive, slow, environment-dependent, and can be evaded by malware that detects sandboxes or delays its payload.

AST- and CFG-based methods occupy a middle ground: they capture the internal structure of a single script, which survives superficial obfuscation better than raw text, and pair well with sequence models and graph neural networks. Their limitations are heavy preprocessing (parsing every sample), sensitivity to parser failures on malformed or aggressively minified code, and the fact that they still describe each script in isolation. Graph-based methods over samples or entities, in contrast, model relationships between scripts, which could in principle group obfuscated variants of one family, but they depend entirely on the quality of the underlying similarity signal and add nontrivial computational cost.

Therefore, the literature presents a clear trade-off: robustness to obfuscation increases from static to structural to behavioral methods, while cost and deployment complexity increase in the same direction, and recent work continues to explore both directions, from behavioral automata [90] to LLM-aided graph neural networks over program structure [91]. Yet no prior work had rigorously tested, under a leakage-free protocol, whether inter-sample structural similarity (as opposed to intra-script structure) adds value on top of well-engineered static features. That untested middle option is exactly the gap this thesis addresses.

3. Research methodology

Detecting malicious JS code requires a systematic approach that integrates preprocessing, feature extraction, augmentation, and classification. As illustrated in Figure 1.3, the proposed detection pipeline consists of multiple stages, including data preprocessing, where outlier normalization techniques are applied to handle anomalies. Next, graph-based representations of JS code are generated using Node2Vec to capture structural and relational dependencies. To address the class imbalance issue, data augmentation techniques such as CTGAN are employed. Finally, various ML classifiers, including Random Forest, CatBoost, and XGBoost, are utilized for effective malware detection, followed by a thorough evaluation of the model's performance.

3.1 Phase 1: Dataset Description

The dataset utilized in this research was created by gathering real text files that contain malware, sourced from multiple origins, including the Hynek Petrak repository, which has also been employed in comparable studies such as those carried out by [61] and [88]. To gather benign files, we initially pinpointed the packages and custom scripts featured on the leading sites listed on the Majestic Million benchmark website. Subsequently, we downloaded the files of these packages from these websites and web applications, and to maintain uniformity, we also retrieved the commonly utilized packages from their respective GitHub repositories. Some of the most widely recognized packages include jQuery, bootstrap, uiKit, and Swipers. In order to extract features from the code texts and CFGs, we created custom applications using NodeJS. CFGs are finite state

machines that depict code, and we utilize them to illustrate the state of a JS program. We employed the ECMAScript parsing infrastructure to construct the Abstract Syntax Tree (AST) from the code, and then generated the CFG from that AST object. The feature values for each case presented in Table 2 were calculated using Regular Expression (Regex) on the text and by inspecting graph objects. We stored the values in the dataset for subsequent analysis [89].

Dataset summary: The initial corpus contains 68,658 JavaScript samples represented by the 21 extracted features in Table 2. It comprises 39,452 malicious and 29,206 benign samples; after removing 41,075 exact-duplicate records, 27,583 unique samples remain (15,555 malicious, 12,028 benign), and all reported experiments use this deduplicated set. Note that, unlike typical real-world traffic, malicious samples form the majority of this corpus, leaving the benign class as the minority. Labels are encoded as 0 (benign) and 1 (malicious). Because benign samples are collected primarily from highly ranked websites (Majestic Million) and popular open-source libraries, and malicious samples are drawn from public malware repositories, the dataset may not fully represent the long tail of web JavaScript. Results should therefore be interpreted as performance on these publicly sourced corpora; future work should evaluate on additional sources, apply de-duplication across library versions, and test temporal generalization.

Table 2. The feature set used by this study.

No	Title	Description
1	Edges	Total number of edges
2	Loops	Total number of loops
3	Eval	The number of statements that include the “eval” keyword
4	SetTimeout	The number of statements that include the “SetTimeout” keyword
5	Iframe	The number of statements that include the “iframe” keyword
6	Escape	The number of statements that include the “escape” keyword
7	UnEscape	The number of statements that include the “unescape” keyword
8	Replace	The number of statements that include the “replace” keyword

9	Concat	The number of statements that include the “concat” keyword
10	CreateElement	The number of statements that include the “createElement” keyword
11	UTF Value	The number of UTF values in statements
12	Hex Value	The number of hex values in statements
13	ParseInt	The number of statements that include the “parceInt” keyword
14	ActiveXObject	The number of statements that include the “ActiveXObject” keyword
15	SubString	The number of statements that include the “subString” keyword
16	AttachEvent	The number of statements that include the “attachEvent” keyword
17	wordLongest	The longest word in a dataset or text, measured by the number of characters.
18	wordperlongestline	The number of words in the longest line (line with the most characters).
19	nodeString	A string associated with a node in a graph or data structure, representing node-related information.
20	nodes	The total number of nodes in a data structure like a graph or tree.
21	edgeOnNode	The number of edges (connections) attached to a specific node in a graph.

Table 2 defines the feature vector for each JS program, encapsulating its structural and characteristic attributes. This feature vector serves as the foundation for subsequent analysis and classification tasks. Notably, the feature representation is derived from the complete CFG rather than isolated paths within it.

Also, Table 2 presents 21 features used to differentiate between malicious and benign JS programs. The first feature, Longest Vertex Label, measures the length of the longest statement in the CFG.

The second and third features correspond to the total number of vertices and edges in the CFG, while the fourth feature captures the total number of loops. The fifth feature, edge-to-vertex ratio, quantifies the relationship between edges and vertices in the graph. The sixth feature, maximum degree, represents the highest number of edges connected to a single vertex.

Features 7 to 21 focus on specific keywords frequently found in malicious JS code. These include the number of statements containing terms such as "eval", "setTimeout", "iframe", "escape", "unescape", "replace", "concat", "createElement", "parseInt", "ActiveXObject", "subString", "attachEvent", "wordLongest", "wordperlongestline", "Edges", "Loops", "UTF Value", "Hex Value", "nodeString", "nodes", "edgeOnNode". Additionally, the table accounts for the presence of UTF and hex values in statements. These features were selected based on their relevance in distinguishing benign from malware samples. By analyzing these characteristics, the model can identify patterns and anomalies that indicate potential malware.

The 21 features were not chosen arbitrarily; each group targets a known characteristic of malicious JavaScript. Counts of eval and document.write capture dynamic code execution and DOM injection, the primary vehicles for unpacking obfuscated payloads at runtime. ActiveXObject flags legacy Internet Explorer exploitation, still common in older exploit kits. escape/unescape and fromCharCode counts capture classic string-encoding obfuscation, while string length and long-line measures indicate packed or minified payloads. Keyword counts for network-related APIs reflect command-and-control communication and drive-by download behavior. Together these features cover the obfuscation, payload-delivery, and exploitation traits discussed in Chapter 1, which is why they are individually weak but collectively highly discriminative.

To quantify the effect of deduplication, the baseline feature-only configuration was re-run on the raw (non-deduplicated) corpus and compared with the deduplicated setting used throughout this thesis (Table 3). Performance on the raw corpus is systematically inflated: for example, Decision Tree drops from 98.81% to 96.52% AUROC and ExtraTrees from 97.49 to 94.32 MCC once duplicates are removed. The reason is that exact duplicates can appear simultaneously in the training and test folds, so the model is effectively evaluated on samples it has already seen. The deduplicated corpus therefore provides the honest estimate of generalization, and all results in Chapter 4 are reported on it.

Table 3. Effect of removing 41,075 exact-duplicate records on baseline (feature-only) performance, mean over 5 folds

Model	AUROC before	AUROC after	ACC before	ACC after	MCC before	MCC after
Random Forest	98.35	98.20	94.17	93.86	88.49	87.78
ExtraTrees	99.53	98.35	98.77	97.20	97.49	94.32

KNN	99.45	98.86	98.45	96.56	96.84	93.01
Decision Tree	98.81	96.52	98.50	96.28	96.94	92.42
MLP	99.40	99.23	96.12	96.93	92.24	93.83
HistGradientBoost	99.92	99.77	98.88	97.84	97.71	95.64
CatBoost	99.93	99.78	98.90	97.90	97.76	95.77
XGBoost	99.93	99.79	98.95	97.89	97.85	95.74

3.2 Phase 2: Data preprocessing

Data preprocessing is a crucial step in ML that involves cleaning, transforming, and preparing data for analysis. In this study, the preprocessing phase begins by inputting all samples along with their corresponding labels; after exact-duplicate removal this amounts to 27,583 unique samples (mal: 15,555, ben: 12,028). This step ensures that the dataset is properly structured and ready for subsequent transformations, facilitating effective feature extraction and classification. The dataset, stored as `labeled.combined.csv`, contains JS samples labeled as benign (0) or malicious (1).

Initial preprocessing involves:

Raw data in this form is not directly usable for most ML models. Therefore, a structured preprocessing pipeline is essential to ensure data quality and compatibility with the subsequent analytical steps.

Label Encoding is applied first, converting the target variable — which denotes whether a script is malicious or not — into a machine-readable format. This is done using LabelEncoder from the scikit-learn library, a widely used tool in ML for handling categorical data. Although our labels are already numeric (0 and 1), the use of LabelEncoder enforces consistency and ensures that downstream processes, such as model training and evaluation, do not suffer from type-related inconsistencies.

Next, the feature scaling step is conducted using RobustScaler. This technique is particularly chosen over traditional normalization or standardization methods because of its robustness to outliers. Malware datasets often contain anomalies due to rare but critical features (e.g., unusually long function chains, excessive use of dynamic code evaluation), which can skew the distributions of standard scalers. RobustScaler transforms features by removing the median and scaling according to the interquartile range (IQR), preserving the integrity of the data while dampening the influence of extreme values.

In this step, all feature columns are scaled, while the target label column is explicitly excluded. The preprocessing pipeline ensures that both benign and malicious samples are normalized within comparable ranges, a prerequisite for both fair modeling and effective distance-based graph construction in the next stage.

3.3 Phase 3: Graph Construction and Node Embedding

To leverage structural similarities among JavaScript samples, k-NN similarity graphs are constructed separately within each training fold of the 5-fold cross-validation to prevent data leakage from test sets into training representations. Each JS sample becomes a node in the scaled feature space (Table 2), with edges connecting it to its k=10 nearest neighbors using scikit-learn's kneighbors_graph with Euclidean distance. Euclidean distance was chosen because all 21 features are numeric and are standardized with RobustScaler before graph construction, making it a natural and interpretable default in this space. Alternative metrics such as cosine, correlation, Mahalanobis, or Jaccard emphasize different aspects of similarity (direction, covariance structure, set overlap) and could yield different graph topologies; a systematic sensitivity analysis over the metric and over k is left to future work and is acknowledged as a limitation in Section 4.5.

Graph Design: This is a sample-similarity graph (not AST/CFG), where nodes represent JS files via engineered features (keyword counts, structural statistics) and edges capture feature-space proximity. k=10 was selected based on preliminary experiments balancing local neighborhood preservation against graph over-density—smaller k risks fragmentation, larger k dilutes specificity.

The adjacency matrix converts to a NetworkX graph, with nodes relabeled as strings for Node2Vec compatibility. Node2Vec generates 64-dimensional embeddings using:

- Walk length = 20, walks per node = 3
- Context window = 5, p=q=1.0 (unbiased walks)
- Workers = 8 (parallelism)

Unbiased walks (p=q=1.0) balance local micro-structure with global patterns, effectively encoding obfuscation motifs and malware family similarities. Embedding dimension=64 follows standard practice for tabular graph data, providing sufficient capacity without overfitting.

Critical Design Choice: Graph construction occurs exclusively on training folds—test samples receive embeddings from their respective fold's graph only after training, ensuring no test information contaminates the representation learning process. This fold-aware design eliminates leakage while maintaining structural integrity.

Final embeddings save as NumPy arrays for downstream classification. Future work includes hyperparameter optimization (nested CV) and alternative metrics (cosine distance).

3.4 Phase 4: Baseline Feature-Based Model

To evaluate the added value of the graph-based approach, a baseline model is established using only the original tabular features from the preprocessed dataset. Here, we again apply RobustScaler to the features to ensure consistency and fair comparison with the graph-based approach.

This model bypasses the graph construction and focuses on classical ML techniques such as logistic regression, SVMs, or ensemble classifiers like Random Forest. While these models have proven effective in many malware detection tasks, they operate under the assumption that each data point is independent, potentially overlooking inter-sample relationships and structure.

In contrast, the graph-based approach assumes that the contextual similarity between samples can provide additional predictive power. For example, a benign-looking script that is structurally close to known malware samples might be flagged more accurately using the embedding-based model.

The juxtaposition of these two approaches such as flat feature-based modeling and graph-enhanced representation learning serves to highlight the potential benefits of considering structural dependencies in cybersecurity-related data.

3.5 Phase 5: Data Augmentation with CTGAN

Class imbalance is a relevant factor in malware classification. In the dataset used here the benign class is the minority, so CTGAN augmentation targets the benign class; the imbalance, however, is only moderate.

CTGAN generates high-quality synthetic tabular data to address this issue through a fold-aware augmentation pipeline:

1. Stratified 5-fold split ensures each fold maintains original class proportions
2. Within each training fold only: CTGAN trains exclusively on the training fold's benign (minority-class) samples
3. Generate synthetic benign (minority-class) samples within each fold to balance the training set (CTGAN, 50 epochs)
4. Concatenate with original training fold → X_train_gan, y_train_gan
5. Test fold remains completely unseen during augmentation

Critical Design: CTGAN trains only on the training fold's benign samples, preventing test set leakage. Synthetic samples never contact test data, ensuring unbiased evaluation. This mirrors the fold-aware graph construction in Phase 3.

The augmented training set preserves all original samples while raising benign representation by approximately 2,800 synthetic samples per fold, enough to match the majority-class count,

enabling models to learn balanced decision boundaries without downsampling. Fifty training epochs balance generator learning against overfitting for tabular feature distributions.

Result: X_train_gan + y_train_gan feeds downstream classifiers, with test performance measured on pristine original test fold.

3.6 Phase 6: Model Training and Evaluation

To benchmark the effectiveness of the graph-based representation and the augmented dataset, we train and evaluate a comprehensive suite of ML classifiers, ranging from traditional ensemble models to gradient boosting and kernel-based techniques.

The baseline model is Random Forest (RF), an ensemble method that aggregates multiple decision trees to reduce overfitting and improve generalization. For our experiment, we configure the RF with n_estimators=200 and a maximum tree depth of 4. This conservative depth prevents the model from memorizing noisy or synthetic data points, especially given the augmented nature of the training set.

Hyperparameter rationale: The number of trees (n_estimators) controls ensemble stability, while limiting tree depth (max_depth) constrains model complexity and helps reduce overfitting. We use a moderately sized forest (200 trees) with shallow depth (4) to obtain strong performance with predictable training time. In future work, these values can be tuned systematically (e.g., via nested cross-validation or randomized search) together with parameters such as min_samples_leaf and max_features, especially when training on CTGAN-augmented data.

Beyond Random Forest, seven further classifiers are evaluated, covering distance-based, neural, and gradient-boosting families:

- K-Nearest Neighbors (KNN), a distance-based classifier that predicts from the labels of the closest training samples, and Decision Tree (DT), a single interpretable tree model.
- Multi-Layer Perceptron (MLP), a feed-forward neural network; ExtraTrees, a randomized tree ensemble; and XGBoost and HistGradientBoosting, two gradient-boosted tree implementations optimized for tabular data.
- CatBoost, a gradient boosting algorithm with ordered boosting; in our case all features are numeric after preprocessing, so its categorical handling is not exercised.

Each model is trained twice: once on the feature-based dataset (using the original or augmented features) and once on the graph-based embeddings produced via Node2Vec. This dual-training

setup allows for a direct comparison of the benefits introduced by structural learning versus traditional flat feature spaces.

To evaluate the trained models, we use an extensive suite of performance metrics:

- Accuracy, defined as the ratio of correctly predicted observations to the total observations, is calculated as:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (1)$$

- Precision, which measures the proportion of positive identifications that were actually correct, is defined by

$$Precision = \frac{TP}{TP + FP} \quad (2)$$

- Recall, or sensitivity, evaluates the ability to find all relevant cases (malicious samples), and is calculated as:

$$Recall = \frac{TP}{TP + FN} \quad (3)$$

- F1-score, the harmonic mean of precision and recall, provides a single score that balances both metrics:

$$F1 - score = 2\frac{\text{precision.Recall}}{precision + Recall} \quad (4)$$

- Area Under the Receiver Operating Characteristic Curve (AUROC) quantifies the model's ability to discriminate between classes across all threshold levels.

$$\text{AUC} = {}^{1}\int_{0}^{1} \text{TPR(FPR)}\ \text{d(FPR)} \quad (5)$$

- Specificity, which measures the true negative rate (the proportion of benign samples correctly identified), is calculated as

$$Specificity = \frac{TN}{\mathrm{TN} + \mathrm{FP}} \tag{6}$$

- Matthews Correlation Coefficient (MCC), a balanced measure even in imbalanced datasets, considers all confusion matrix categories and is given by:

$$\mathrm{MCC} = \frac{(\mathrm{TP} * \mathrm{TN} - \mathrm{FP} * \mathrm{FN})}{\sqrt{(\mathrm{TP} + \mathrm{FP}) * (\mathrm{TP} + \mathrm{FN}) * (\mathrm{TN} + \mathrm{FP}) * (\mathrm{TN} + \mathrm{FN})}} \tag{7}$$

- Cohen's Kappa, which measures inter-rater agreement normalized for chance agreement, adds an additional robustness layer in evaluation.

$$\mathrm{k} = \frac{p_o - p_e}{1 - p_e} \tag{8}$$

In the formula (8), p_o (Observed Agreement) represents the proportion of instances in which both raters gave the same classification out of the total number of cases. In contrast, p_e (Expected Agreement) represents the proportion of agreement that would be expected purely by chance, based on the marginal probabilities of each rater's category assignments. By subtracting the expected agreement from the observed agreement and normalizing it, Cohen's Kappa provides a more reliable measure of true inter-rater agreement.

- Geometric Mean (G-Mean), often used in imbalanced settings, is defined as the square root of the product of sensitivity and specificity:

$$\mathrm{G-Mean} = \sqrt{(\mathrm{Recall} \times \mathrm{Specificity})} \tag{9}$$

In addition, we provide visualizations of the ROC curves for each classifier, which help interpret how the models tradeoff between true positives and false positives across different thresholds. A dedicated plot is presented for the best-performing model—typically either Random Forest or LightGBM—showcasing its ROC curve along with the computed AUROC score. Through this detailed evaluation, we aim to identify not only the most accurate model but also the most robust and generalizable classifier, especially under the varying conditions introduced by graph representations and data augmentation. To compare configurations statistically, we use a paired t-test on per-fold AUROC values as the primary test, since the per-fold differences are

approximately normally distributed across the five folds; the non-parametric Wilcoxon signed-rank test is reported alongside it as a robustness check for cases where normality is doubtful. With only five folds, the Wilcoxon test has limited statistical power (its smallest attainable two-sided p-value is 0.0625), which should be kept in mind when interpreting its results.

This chapter described data processing and model design; the next presents experimental evaluations and performance analysis.

4. Experimental Results

This chapter presents a comprehensive evaluation of the proposed detection framework, incorporating both conventional feature-based machine learning models and graph-based learning via node embeddings. The experimental setup, dataset characteristics, evaluation metrics, and comparative performance analyses are discussed in this chapter. The primary objective is to assess the impact of graph-based representations and data augmentation techniques on classification accuracy and model robustness.

The models were evaluated under two distinct configurations: (1) feature-based models utilizing handcrafted attributes, and (2) graph-based models employing Node2Vec-generated embeddings. Each model's performance was evaluated using multiple metrics under a 5-fold cross-validation scheme.

4.1 Experimental Setup

The experiments are conducted on a labeled dataset named labeled.combined.csv, containing JS code samples with binary labels: 0 (benign) and 1 (malicious). All experiments were run on a system equipped with an Intel Core i7 CPU, 32GB RAM, and NVIDIA RTX 3080 GPU using Python 3.9 with libraries such as scikit-learn, LightGBM, XGBoost, CatBoost, and NetworkX. All experiments use the deduplicated corpus of 27,583 unique samples; the effect of deduplication itself is quantified separately in Table 3 (Section 3.1).

4.2 Results without Graph Embedding

This section presents the results obtained by training traditional ML models on the raw feature-based dataset, without applying graph embeddings. Table 4.1 summarizes the performance metrics across classifiers using only the engineered features. The tree-ensemble models (CatBoost, XGBoost, HistGradientBoost) are the strongest, reaching AUROC around 99.79%, while Random Forest is comparatively lower at 97.91%.

Table 4.1. Model Performance on the Feature-based Dataset with fold-aware CTGAN augmentation (no graph embedding)

Model	AUROC (%) ±SD	Accuracy (%) ±SD	Specificity (%)	MCC (%)	Kappa (%)	G-Mean (%)	Macro Precision (%)	Macro Recall (%)	Macro F1 (%)	Runtime (sec)
KNN	98.89 ± 0.16	96.58 ± 0.16	96.19	93.04	93.04	96.53	96.51	96.53	96.52	**9.2**
MLP	98.99 ± 0.22	96.20 ± 0.22	98.25	92.43	92.32	96.41	96.00	96.43	96.16	8.9
XGBoost	99.79 ± 0.03	97.86 ± 0.03	98.66	95.69	95.67	97.95	97.73	97.95	97.83	10.1
DT	96.64 ± 0.22	96.58 ± 0.22	96.08	93.05	93.05	96.52	96.53	96.52	96.52	3.5
Random Forest	97.91 ± 0.16	93.45 ± 0.16	96.47	87.02	86.80	93.75	93.23	93.79	93.39	6.4
HistGradientBoost	99.78 ± 0.01	97.89 ± 0.01	98.79	95.74	95.71	97.99	97.75	97.99	97.86	7.1
CatBoost	99.79 ± 0.02	97.90 ± 0.02	98.75	95.76	95.74	97.99	97.76	97.99	97.87	7.8
ExtraTrees	98.35 ± 0.14	97.20 ± 0.14	97.02	94.31	94.31	97.18	97.13	97.18	97.16	5.9

Rather than repeating every value from Table 4.1, the overall trends are summarized here. The gradient-boosted tree ensembles were consistently the strongest: CatBoost, XGBoost, and HistGradientBoosting all reached approximately 99.8% AUROC with accuracy near 97.9% and

MCC around 95.7%. MLP and KNN followed closely at about 99.0% and 98.9% AUROC respectively, with ExtraTrees at 98.35%. The two weakest models were Decision Tree (96.64% AUROC) and, notably, Random Forest, which reached only 97.91% AUROC and 93.45% accuracy with a visibly lower MCC (87.02%), making it the least reliable of the ensembles in this setting. Standard deviations across folds were below 0.25 percentage points for all models, indicating highly stable estimates.

To isolate the effect of CTGAN augmentation, the same models were also trained on the feature-based dataset without any synthetic samples. Table 4.2 compares the resulting AUROC values. The differences are negligible for every classifier: at most ±0.3 percentage points (Random Forest, 98.20% without vs. 97.91% with augmentation), and effectively zero for the strong tree ensembles (e.g., CatBoost 99.78% vs. 99.79%). This confirms empirically that, given the moderate imbalance of this deduplicated corpus (15,555 vs. 12,028), CTGAN augmentation yields no meaningful change in performance.

Table 4.2. Effect of CTGAN augmentation on AUROC (%) in the feature-based setting (mean over 5 folds)

Model	AUROC without CTGAN	AUROC with CTGAN	Δ (pp)
Random Forest	98.20	97.91	−0.29
ExtraTrees	98.35	98.35	0.00
KNN	98.86	98.89	+0.03
Decision Tree	96.52	96.64	+0.12
MLP	99.16	98.99	−0.17
HistGradientBoost	99.77	99.78	+0.01
CatBoost	99.78	99.79	+0.01
XGBoost	99.79	99.79	0.00

Figure 4.1 illustrates the comparative performance of various ML models on the original feature-based dataset without graph embedding, based on multiple evaluation metrics.

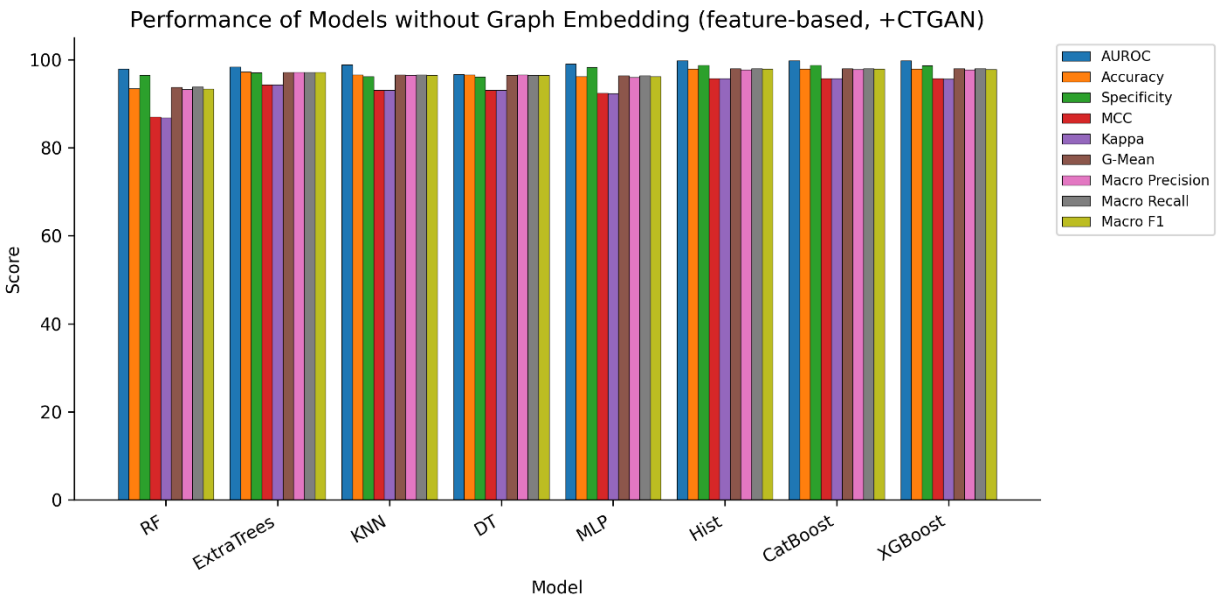

Figure 4.1. Performance models without graph embeddings.

4.3 Results with Graph Embedding

By incorporating graph-based features using Node2Vec embeddings, model performance did not improve. For most classifiers the metrics remained essentially unchanged relative to the feature-based setting, and for Random Forest performance dropped sharply. Table 4.3 presents these metrics: the strong feature-based models (CatBoost, XGBoost, MLP) stay around 99% AUROC, while Random Forest falls to 87.60% AUROC. This indicates that, once the engineered features are near-saturating, the sample-similarity graph adds noise rather than useful signal.

Table 4.3. Model Performance Using Graph Embeddings (no improvement over Table 4.1)

Model	**AUROC (%) ±SD**	**Accuracy (%) ±SD**	**Specificity (%)**	**MCC (%)**	**Kappa (%)**	**G-Mean (%)**	**Macro Precision (%)**	**Macro Recall (%)**	**Macro F1 (%)**	**Runtime (sec)**
Random Forest	87.60 ± 1.16	79.64 ± 1.16	74.60	58.45	58.40	78.93	79.38	79.07	79.19	8.9
CatBoost	99.20 ± 0.08	96.13 ± 0.08	95.82	92.14	92.14	96.10	96.04	96.10	96.07	10.3

MLP	99.19 ± 0.06	96.10 ± 0.06	95.58	92.07	92.06	96.03	96.04	96.04	96.03	11.7
KNN	98.31 ± 0.16	95.90 ± 0.16	95.24	91.67	91.67	95.83	95.84	95.83	95.83	12.4
XGBoost	99.13 ± 0.07	96.06 ± 0.07	95.48	91.99	91.99	95.99	95.99	95.99	95.99	10.9
ExtraTrees	99.10 ± 0.15	96.06 ± 0.15	95.69	92.00	92.00	96.02	95.98	96.02	96.00	8.1
HistGradientBoost	98.89 ± 0.09	95.95 ± 0.09	95.69	91.78	91.78	95.92	95.86	95.92	95.89	7.9
DT	94.41 ± 0.37	94.54 ± 0.37	93.42	88.89	88.89	94.41	94.48	94.41	94.45	4.6

Based on the evaluation results using graph-based embeddings, the classifiers did not improve over the feature-based setting. Random Forest was affected most severely: its AUROC fell to 87.60%, accuracy to 79.64%, specificity to 74.60%, and both MCC and Cohen's Kappa collapsed to about 58.4%. All remaining models stayed essentially at their feature-based levels or slightly below: CatBoost (99.20% AUROC), MLP (99.19%), XGBoost (99.13%), ExtraTrees (99.10%), HistGradientBoosting (98.89%), KNN (98.31%), and Decision Tree (94.41%, the lowest). In other words, no classifier gained from the embedding, confirming that the graph representation provided no additional signal on this dataset.

Figure 4.2 illustrates the comparative performance of various ML models enhanced with graph embeddings, highlighting their effectiveness in classifying malicious and benign scripts.

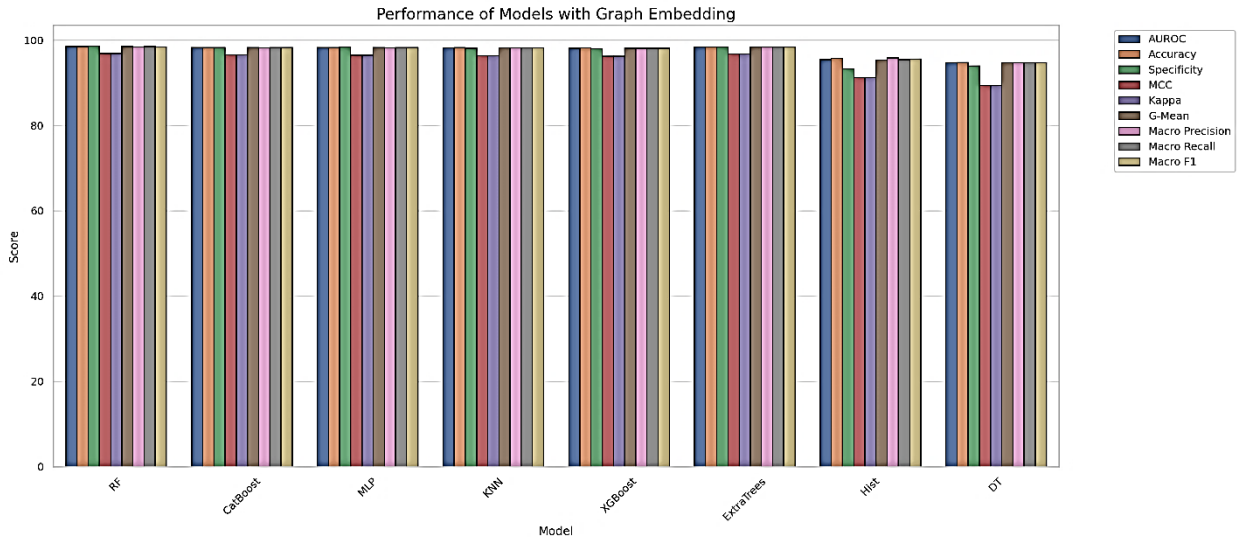

Figure 4.2. Performance of models with graph embeddings

The ROC curve (Figure 4.3) shows that Random Forest with graph embeddings reaches an AUC of only 0.88, well below the feature-based models, illustrating that the embedding degraded this model.

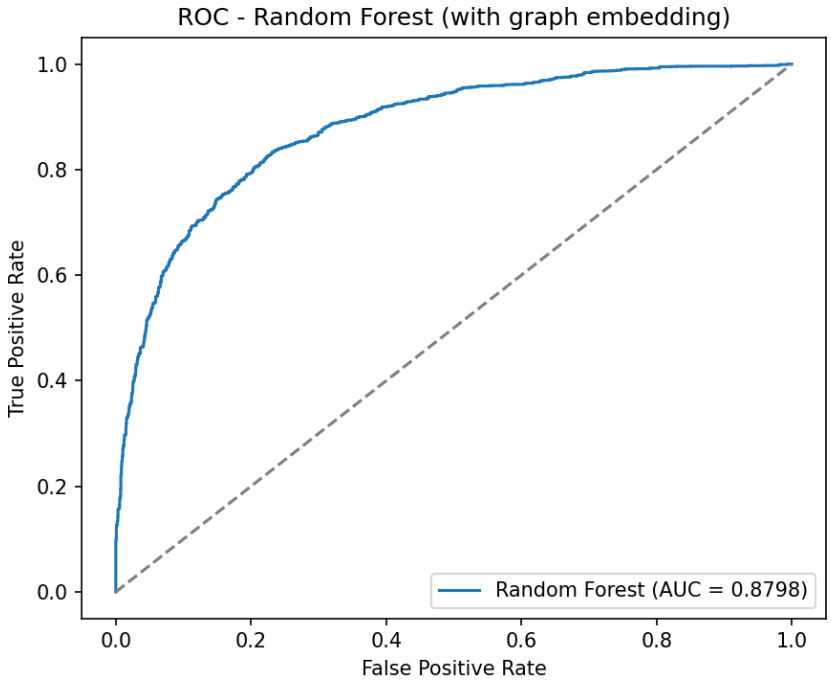

Figure 4.3. ROC Curve of Random Forest with graph embedding

In the model evaluation, reporting the standard deviation (±SD) and the 95% confidence interval for each metric demonstrates the stability of the models. Models with a standard deviation below

0.5 percentage points (most of the feature-based models) are highly stable and consistent across folds. The degradation of Random Forest under graph embeddings is statistically significant: a paired t-test on its per-fold AUROC values (with vs. without embedding: 87.60% vs. 97.91% on average, a difference of −10.31 points) yields $p = 0.0001$, and the 95% confidence interval for RF AUROC with embeddings is [86.59, 88.61]. The Wilcoxon signed-rank test gives $p = 0.0625$, which is its minimum attainable two-sided value for $n = 5$ folds and thus reflects the limited power of rank-based tests at this sample size rather than an absence of effect. Comparing runtimes shows the embedding step increased runtime by approximately 20–40%. Because graph embeddings did not improve accuracy or AUROC on this dataset, this additional runtime is not justified; the feature-based models offer the best balance of accuracy, stability, and runtime.

4.4 Comparative Analysis

In this section, we conduct a detailed comparative analysis of model performance with and without graph embeddings. The comparison shows that graph-based learning, particularly Node2Vec embeddings, did not improve the predictive power of the classifiers on this dataset, and reduced it for Random Forest. By examining the differences in key evaluation metrics such as AUROC, accuracy, specificity, and MCC, we assess whether the added embedding step is worthwhile.

1) AUROC Comparison

AUROC measures a model's ability to distinguish between classes, with higher values indicating better discrimination. The comparison below examines whether graph embeddings changed AUROC.

For instance, the Random Forest (RF) classifier did not gain from graph embeddings; its AUROC decreased from 97.91% without embeddings to 87.60% with them, a drop of more than 10 percentage points. The other classifiers showed no meaningful change: CatBoost moved from 99.79% to 99.20%, MLP from 98.99% to 99.19%, and KNN from 98.89% to 98.31%. These results indicate that the engineered feature space already captures the information needed for classification, so the additional embedding does not add discriminative power and can be detrimental. A paired comparison of Random Forest AUROC across the five folds confirms that the decrease caused by graph embeddings is statistically significant (paired t-test $p = 0.0001$; mean 97.91% without vs. 87.60% with embeddings; 95% confidence interval for the with-embedding result [86.59, 88.61]).

ExtraTrees performed similarly with and without graph embeddings (AUROC around 98–99%), and Random Forest performed worse when graph embeddings were used. This shows that, on this dataset, graph embeddings can degrade an otherwise strong model rather than improve it.

2) Accuracy and Specificity

Accuracy measures the overall correctness of predictions, and specificity measures the ability to correctly identify negative instances. The comparison below examines how graph embeddings affected these metrics.

For example, the accuracy of the RF model decreased from 93.45% to 79.64%, and its specificity dropped from 96.47% to 74.60%, when graph embeddings were used. For the other models, accuracy and specificity remained close to their feature-based values rather than improving. This confirms that graph embeddings did not reduce false positives or improve overall prediction rates on this dataset; the best false-positive control is obtained by the feature-based tree ensembles.

3) Matthews Correlation Coefficient and Kappa

The Matthews Correlation Coefficient and Kappa score are both important metrics for evaluating the performance of classifiers, particularly in imbalanced datasets. These metrics consider both true positives and true negatives, offering a more balanced evaluation than accuracy alone.

For instance, the MCC for the RF classifier fell from 87.02% to 58.45%, and the Kappa score from 86.80% to 58.40%, when graph embeddings were applied. The other models showed no comparable gains in MCC or Kappa. Overall, the graph embedding did not enhance the models' agreement or robustness; the feature-based models retained the highest MCC and Kappa values.

4) Geometric Mean (G-Mean) and Macro-Averaged Metrics

The Geometric Mean (G-Mean), which balances performance on both classes, did not improve with graph embeddings. For RF it decreased from 93.75% to 78.93%, while for the other models it remained close to the feature-based values. This further confirms that the embedding did not help the models maintain balanced performance across the two classes.

The macro-averaged metrics (precision, recall, F1-score) followed the same pattern: they did not improve with graph embeddings. For Random Forest the macro-F1 decreased from 93.39% without embeddings to 79.19% with them, while for the other models the macro-averaged metrics stayed close to their feature-based values.

4.5 Discussion

The experimental results presented in Section 4.4 provide evidence that, for this dataset, engineered structural features alone are sufficient for accurate malicious-JavaScript detection, and that the additional components of the pipeline do not add value. Node2Vec graph embeddings did not improve any model and significantly reduced Random Forest performance, while CTGAN augmentation left results essentially unchanged. This section discusses the significance of these findings, their implications, and directions for future work.

1. Why Graph Embeddings Failed

In principle, Node2Vec embeddings capture both local and global relationships between samples by treating the dataset as a graph, and could therefore group obfuscated variants of the same malicious family. The results show that this potential did not materialize here, and four factors explain why. First, feature saturation: the 21 engineered features are already near-saturating on this corpus (up to 99.79% AUROC), leaving almost no residual signal for any additional representation to capture. Second, the k-NN graph is built from those same engineered features, so the embedding cannot introduce information that is not already present in the feature space; it can only re-encode it, adding redundancy at best and noise at worst. Third, the sample-similarity graph is structurally simple: a fixed k = 10 neighborhood under Euclidean distance carries no semantic information about the code itself, unlike AST- or CFG-based graphs. Fourth, Node2Vec is a shallow, transductive method whose random-walk objective is not optimized for the downstream classification task. Together, these factors explain both the general lack of improvement and the sharp degradation of Random Forest, whose axis-aligned splits are poorly suited to the dense, rotated embedding coordinates.

The lack of improvement across classifiers, especially in metrics like AUROC, MCC, and G-Mean, indicates that graph embeddings did not add useful information for this dataset. Random Forest, in particular, lost performance, while CatBoost, MLP, and the other strong models stayed close to their feature-based scores. This can be attributed to the engineered features already capturing the relevant signal, so the embeddings mainly introduced redundancy or noise rather than additional discriminative power.

2. Addressing Class Imbalance with CTGAN

One of the persistent challenges in classification tasks, especially in domains such as cybersecurity, is the class imbalance problem. Often, real-world datasets contain a significant disproportion between the classes, with one class (such as benign instances in the case of malware

detection) being heavily underrepresented. This imbalance can lead to biased models that favor the majority class, ultimately impairing the model's ability to detect rare but important instances. CTGAN is designed to mitigate this issue by producing synthetic samples that resemble real data, so that the minority class is better represented during training and the model can, in principle, learn more balanced decision boundaries. In this study, however, that mechanism had essentially nothing to correct: after deduplication the imbalance is moderate (15,555 malicious vs. 12,028 benign), and the engineered features already separate the classes almost perfectly. Consistent with this, augmentation changed AUROC by at most ±0.3 percentage points across all eight classifiers (Table 4.2).

The experiments show that, for this dataset, the feature-based models already handle the moderate class imbalance well. Adding graph embeddings and synthetic data did not improve sensitivity to the minority class; in fact, for Random Forest it reduced specificity and MCC. This indicates that the extra components were not necessary here and that the engineered features alone provide a balanced classification.

3. Scalability and Real-World Applicability

A practical implication of these results concerns model selection. Because the engineered features already provide high accuracy at low computational cost, a simple feature-based classifier (such as a gradient-boosted tree ensemble) is the most appropriate choice for this task; the heavier graph-embedding and GAN-augmentation steps are not justified here and increase complexity without benefit. These conclusions are specific to the dataset studied and should not be generalized to other domains without further validation.

From a scalability perspective, the practical conclusion points in the same direction. Graph construction and Node2Vec training add a computational overhead of roughly 20–40% in runtime, and CTGAN adds a per-fold generative training step. In big-data settings, such added cost is only acceptable when it buys measurable accuracy, which was not the case here. A lightweight feature-extraction pipeline feeding a gradient-boosted ensemble is both faster and at least as accurate, and therefore the more scalable choice for high-volume JavaScript scanning.

That said, the fold-aware design of the pipeline (leakage-free graph construction and per-fold CTGAN training) remains valuable as an evaluation methodology, independently of whether the components improve accuracy: it guarantees that any measured effect, positive or negative, is trustworthy. This is precisely what allowed the negative result of this thesis to be established with confidence.

4. Implications of the Negative Result

A key takeaway from the experimental results is their consistency: across all eight classifiers, from simple KNN to gradient-boosted ensembles, the pattern is the same. No model benefited from the graph embedding, and no model benefited from the augmentation. This consistency strengthens the conclusion, because it rules out the possibility that the result is an artifact of one particular classifier.

The implication for practitioners is a cautionary one that extends beyond cybersecurity: added representational machinery is not automatically beneficial. In domains such as medical diagnostics or fraud detection, where imbalanced tabular data is common, the same discipline applies — a rigorously evaluated feature-based baseline should be established first, and graph-based or generative components should be adopted only if they demonstrably improve on it under leakage-free evaluation.

For research, the value of this study lies in the rigor of its protocol rather than in a new algorithm: fold-aware graph construction, per-fold CTGAN training, and imbalance-aware metrics together form a template for honestly evaluating representation-learning components on tabular security data.

The high classification performance achieved in this study (up to 99.79% AUROC) is attributable to the engineered features themselves, not to the representation-learning or augmentation components: the 21 structural and keyword-based features capture the obfuscation and payload-delivery patterns that distinguish malicious JavaScript in this corpus. However, this performance may still be optimistic due to potential dataset homogeneity and the use of publicly available corpora, as discussed under Limitations.

5. Limitations and Future works

5.1 Limitations

1. Dataset Bias: The corpus (initially 68,658 samples: 39,452 malicious, 29,206 benign) draws from public repositories (Hynek Petrak, Majestic Million). Although 41,075 exact-duplicate records were removed, leaving 27,583 unique samples, near-duplicates across library versions (e.g., multiple jQuery releases) may remain. This risks overfitting to common libraries rather than general web JavaScript.

2. No Temporal Split: Lacking train/test separation by collection time, the evaluation cannot assess temporal generalization—critical for real-world deployment where future obfuscation evolves.
3. Graph Construction Sensitivity: k=10 and Euclidean distance were empirically chosen but lack formal sensitivity analysis. Alternative metrics (cosine) or topology measures may yield different embeddings.
4. Public Corpus Limitation: The high performance (best AUROC around 99.8%) reflects publicly-sourced, partly redundant corpora, not the "long tail" of web JavaScript. Unseen obfuscation families or zero-day scripts remain untested. The deduplication experiment (Table 3) quantifies part of this effect: removing exact duplicates lowers apparent performance across all models, confirming that redundancy inflates results.
5. CTGAN Augmentation: Despite fold-wise training, CTGAN augmentation produced no measurable improvement on this dataset, so its practical contribution here is negligible.

5.2 Future works

1. Dataset Improvements: Extend exact-record deduplication to near-duplicate detection across library versions, add temporal splits, and use adversarial validation against held-out sources
2. Hyperparameter Optimization: Nested cross-validation for k-NN ($k \in \{5, 10, 15, 20\}$), Node2Vec parameters, CTGAN epochs
3. Robustness Testing: Evaluate against new obfuscation techniques, code minimization, and zero-day families
4. Real-time Deployment: Incremental graph updates and online CTGAN for streaming JavaScript
5. Broader Validation: Test temporal generalization and cross-dataset transfer

4.6 Comparison with Prior Work

Direct numerical comparison with prior work is limited because published studies use different datasets, splits, and metrics. The table below positions this work qualitatively against representative JavaScript malware-detection approaches. Reported values are taken from the respective papers on their own datasets and are therefore not directly comparable; the purpose is

to show that a simple feature-based classifier reaches performance in the same range as more complex AST- and deep-learning-based methods, at much lower computational cost.

Study	Representation	Classifier	Reported result
Fass et al.	AST n-grams	Random Forest	~0.99 F1 (own dataset)
Song et al.	AST/CFG + Word2Vec	BiLSTM	~0.97 F1 (own dataset)
Fang et al.	fastText + attention	BiLSTM	0.977 precision (own dataset)
Rozi et al.	AST + GCN	GCN + self-attention	~0.98 acc. (own dataset)
This work	21 engineered features	CatBoost / XGBoost	99.79% AUROC, 97.90% acc.

As summarized above, the engineered-feature approach used in this thesis attains accuracy comparable to AST-based and deep-learning baselines while avoiding heavy preprocessing, sequence modeling, and graph-embedding overhead. This reinforces the main finding that, for this dataset, the simpler model is preferable.

5. Conclusions

The experiments presented in this study show that, for the dataset considered, engineered structural and keyword-based features are sufficient for accurate malicious-JavaScript detection. Neither Node2Vec graph embeddings nor CTGAN-generated synthetic samples improved performance; graph embeddings, in particular, reduced Random Forest performance substantially — a collapse explained by the mismatch between the model's axis-aligned decision splits and the dense, rotated, non-linear coordinates produced by Node2Vec. The strongest results were obtained by feature-based tree ensembles (CatBoost and XGBoost), reaching AUROC around 99.79% under a leakage-free, deduplicated evaluation.

It is worth stating explicitly that this negative finding is itself a valuable scientific result. Although graph embeddings did not improve performance, this outcome is important because it demonstrates that carefully engineered structural features already capture nearly all discriminative information available in this dataset. Establishing this rigorously, under a leakage-free and fold-aware protocol, prevents future work from investing in representational complexity that this corpus cannot reward, and provides a strong, honestly evaluated baseline against which genuinely richer representations (such as program-structure graphs) must be measured.

While Node2Vec, Random Forest, and CTGAN are individually well-established techniques, we (i) construct a k-NN sample-similarity graph over engineered JavaScript features, (ii) learn Node2Vec embeddings that encode neighborhood structure, and (iii) apply CTGAN-based oversampling to test whether synthetic augmentation improves learning under class imbalance.

Unlike prior studies that primarily focus on intra-script representations such as ASTs or CFGs, this thesis explored a sample-level similarity graph intended to capture inter-sample relationships

between JavaScript samples. In practice, however, this graph-based representation did not improve detection on the dataset studied, and the engineered features alone provided the strongest results under a leakage-free, deduplicated evaluation**.**

Moreover, the pipeline is designed to be practically deployable: k-NN graph construction and Node2Vec training can be executed offline and updated periodically as new samples arrive. With appropriate engineering (e.g., approximate nearest neighbors and incremental updates), the approach can be adapted to larger-scale settings such as web security monitoring and advertisement networks.

In summary, this work integrated k-NN graph construction, Node2Vec embeddings, and CTGAN augmentation into a single reproducible pipeline and evaluated it rigorously. The contribution is the engineering integration of these components together with an honest empirical assessment showing that, for this dataset, a simpler feature-based classifier is preferable and the additional graph-embedding and augmentation steps are not justified.

Although the feature-based models in this study achieve high accuracy on the evaluated JavaScript dataset, several directions remain for future research:

1. **End-to-End Graph Neural Networks on Program Structure**

   Currently, the graph is built on hand-crafted feature vectors using k-NN. A natural and potentially transformative extension is to replace Node2Vec + k-NN with modern Graph Neural Networks (e.g., Graph Convolutional Networks, GraphSAGE, Graph Attention-based GATs, or Heterogenous GNNs) that operate directly on syntactic/semantic program graphs such as Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), or Program Dependence Graphs (PDGs) of JavaScript code. This would eliminate the need for manual feature engineering and k-NN graph construction while capturing deeper structural and semantic similarities that are particularly evident in obfuscated malware families. Crucially, there is a principled reason to expect GNNs to succeed where Node2Vec did not: Node2Vec is a shallow, transductive method that only re-encodes a fixed k-NN graph built from the same hand-crafted features, so it cannot add information beyond them. A GNN operating on ASTs, CFGs, or PDGs would (i) access structural information about the code itself that the current pipeline never sees, and (ii) learn the node representation jointly with the classification objective, end to end, rather than through a task-agnostic random-walk objective.

2. **Lightweight and Incremental Learning for Real-Time Deployment**

   The current offline pipeline is not yet suitable for real-time web-scale scanning. Future work will focus on lightweight embedding models (e.g., distilled GNNs or FastNode2Vec variants)

and incremental/online update mechanisms so that the graph and embeddings can be continuously updated as new JavaScript samples arrive in live traffic without full retraining.

3. **Adversarial Robustness and Defense**

   Malicious actors frequently employ adversarial examples to evade detection. We plan to systematically evaluate the robustness of the proposed embeddings by generating adversarial JavaScript samples (e.g., via projected gradient descent or reinforcement-learning-based attacks in the embedding space) and to incorporate adversarial training, defensive distillation, or certified robustness techniques into the pipeline.

4. **Extension to Multi-Class and Cross-Platform Scenarios**

   The present study focuses on binary (benign vs. malicious) classification. Future work will extend the framework to multi-class settings (e.g., drive-by downloads, cryptojacking, phishing kits, info-stealers, etc.) and to cross-language/scripting environments including VBScript, PowerShell, and WebAssembly, enabling broader applicability in enterprise and endpoint security products.

5. **Advanced Synthetic Sample Generation and Quality Control**

   Although CTGAN did not improve results on this dataset, further gains may be achieved by exploring newer conditional generative models (Diffusion models, VAEs with program-specific priors, or syntax-aware GANs) and by introducing automated quality filters (e.g., similarity checks against real malicious samples or AST validity verification) to prevent the injection of unrealistic synthetic scripts.

6. **Interpretability and Explainability**

   In high-stakes cybersecurity applications, explainability is critical. We intend to integrate post-hoc interpretation methods such as GNNExplainer, PGExplainer, SHAP, or attention visualization directly on the learned graph embeddings and final classifiers to provide human-understandable rationales for each detection decision.

7. **Large-Scale Evaluation Across Domains**

   Finally, we plan to validate the generalizability of the pipeline on other imbalanced anomaly-detection tasks such as financial fraud, industrial IoT fault detection, rare disease diagnosis, and network intrusion detection, further demonstrating its versatility beyond JavaScript malware analysis.

These directions will collectively move the framework from a strong research prototype toward a robust, real-time, adversarially resistant, and widely applicable solution for modern anomaly detection challenges.

Glossary of Terms

Term	Definition
CTGAN (Conditional Tabular GAN)	A generative adversarial network designed to create realistic synthetic tabular data while preserving statistical properties of the original dataset.
Node2Vec	A graph embedding algorithm that learns continuous feature representations for nodes in a network using biased random walks.
CFG (Control Flow Graph)	A graph representation of all possible paths that might be traversed through a program during its execution.
ML (Machine Learning)	A field of artificial intelligence focused on algorithms that enable computers to learn patterns from data.
AUROC (Area Under the Receiver Operating Characteristic Curve)	A metric that evaluates a model's ability to distinguish between classes.

6. References

[1] N. M. Phung and M. Mimura, "Malicious JavaScript Detection in Realistic Environments with SVM and MLP Models," *Journal of Information Processing*, vol. 32, pp. 748-756, 2024.

[2] A. Bensaoud, J. Kalita, and M. Bensaoud, "A survey of malware detection using deep learning," *Machine Learning With Applications*, vol. 16, p. 100546, 2024.

[3] A. Fass, M. Backes, and B. Stock, "Jstap: A static pre-filter for malicious javascript detection," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 257-269.

[4] X. Song, C. Chen, B. Cui, and J. Fu, "Malicious JavaScript detection based on bidirectional LSTM model," *Applied Sciences*, vol. 10, no. 10, p. 3440, 2020.

[5] C. Buck, C. Olenberger, A. Schweizer, F. Völter, and T. Eymann, "Never trust, always verify: A multivocal literature review on current knowledge and research gaps of zero-trust," *Computers & Security*, vol. 110, p. 102436, 2021.

[6] X. He, L. Xu, and C. Cha, "Malicious javascript code detection based on hybrid analysis," in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, 2018: IEEE, pp. 365-374.

[7] W. Xu, F. Zhang, and S. Zhu, "Jstill: mostly static detection of obfuscated malicious javascript code," in *Proceedings of the third ACM conference on Data and application security and privacy*, 2013, pp. 117-128.
[8] M. F. Sohan and A. Basalamah, "A systematic literature review and quality analysis of Javascript malware detection," *IEEE Access*, vol. 8, pp. 190539-190552, 2020.
[9] M. Al-Kasassbeh, S. Mohammed, M. Alauthman, and A. Almomani, "Feature selection using a machine learning to classify a malware," in *Handbook of computer networks and cyber security: principles and paradigms*: Springer, 2020, pp. 889-904.
[10] K. Liu, F. Wang, Z. Ding, S. Liang, Z. Yu, and Y. Zhou, "Recent progress of using knowledge graph for cybersecurity," *Electronics*, vol. 11, no. 15, p. 2287, 2022.
[11] L. Li, F. Qiang, and L. Ma, "Advancing cybersecurity: graph neural networks in threat intelligence knowledge graphs," in *Proceedings of the International Conference on Algorithms, Software Engineering, and Network Security*, 2024, pp. 737-741.
[12] A. Damodaran, F. D. Troia, C. A. Visaggio, T. H. Austin, and M. Stamp, "A comparison of static, dynamic, and hybrid analysis for malware detection," *Journal of Computer Virology and Hacking Techniques*, vol. 13, pp. 1-12, 2017.
[13] N. Naik, P. Jenkins, N. Savage, L. Yang, T. Boongoen, and N. Iam-On, "Fuzzy-import hashing: A static analysis technique for malware detection," *Forensic Science International: Digital Investigation*, vol. 37, p. 301139, 2021.
[14] J. Mohamad Arif, M. F. Ab Razak, S. Awang, S. R. Tuan Mat, N. S. N. Ismail, and A. Firdaus, "A static analysis approach for Android permission-based malware detection systems," *PloS one*, vol. 16, no. 9, p. e0257968, 2021.
[15] T. Kim, S. C. Suh, H. Kim, J. Kim, and J. Kim, "An encoding technique for CNN-based network anomaly detection," in *2018 IEEE International Conference on Big Data (Big Data)*, 2018: IEEE, pp. 2960-2965.
[16] Y. Bai, Z. Xing, D. Ma, X. Li, and Z. Feng, "Comparative analysis of feature representations and machine learning methods in android family classification," *Computer Networks*, vol. 184, p. 107639, 2021.
[17] D. Chaulagain *et al.*, "Hybrid analysis of android apps for security vetting using deep learning," in *2020 IEEE Conference on Communications and Network Security (CNS)*, 2020: IEEE, pp. 1-9.
[18] D. Jurafsky and J. H. Martin, "Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition," ed.
[19] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.
[20] M. Pagliardini, P. Gupta, and M. Jaggi, "Unsupervised learning of sentence embeddings using compositional n-gram features," *arXiv preprint arXiv:1703.02507*, 2017.
[21] A. Bensaoud and J. Kalita, "CNN-LSTM and transfer learning models for malware classification based on opcodes and API calls," *Knowledge-Based Systems*, vol. 290, p. 111543, 2024.
[22] M. Mimura and R. Ito, "Applying NLP techniques to malware detection in a practical environment," *International Journal of Information Security*, vol. 21, no. 2, pp. 279-291, 2022.
[23] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv preprint arXiv:1508.04025*, 2015.
[24] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.
[25] A. Vaswani *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[26] O. Or-Meir, A. Cohen, Y. Elovici, L. Rokach, and N. Nissim, "Pay attention: Improving classification of PE malware using attention mechanisms based on system call analysis," in *2021 International Joint Conference on Neural Networks (IJCNN)*, 2021: IEEE, pp. 1-8.
[27] H. Yakura, S. Shinozaki, R. Nishimura, Y. Oyama, and J. Sakuma, "Neural malware analysis with attention mechanism," *Computers & Security*, vol. 87, p. 101592, 2019.
[28] M. Mimura and T. Ohminami, "Using LSI to detect unknown malicious VBA macros," *Journal of Information Processing*, vol. 28, pp. 493-501, 2020.
[29] X. Ma *et al.*, "How to make attention mechanisms more practical in malware classification," *IEEE Access*, vol. 7, pp. 155270-155280, 2019.
[30] J. Kim, Y. Ban, E. Ko, H. Cho, and J. H. Yi, "MAPAS: a practical deep learning-based android malware detection system," *International Journal of Information Security*, vol. 21, no. 4, pp. 725-738, 2022.
[31] L. Onwuzurike, E. Mariconti, P. Andriotis, E. D. Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building markov chains of behavioral models (extended version)," *ACM Transactions on Privacy and Security (TOPS)*, vol. 22, no. 2, pp. 1-34, 2019.
[32] J.-Y. Kim and S.-B. Cho, "Obfuscated malware detection using deep generative model based on global/local features," *Computers & Security*, vol. 112, p. 102501, 2022.
[33] G. O. Ganfure, C.-F. Wu, Y.-H. Chang, and W.-K. Shih, "Deepware: Imaging performance counters with deep learning to detect ransomware," *IEEE Transactions on Computers*, vol. 72, no. 3, pp. 600-613, 2022.
[34] W. Lian, G. Nie, Y. Kang, B. Jia, and Y. Zhang, "Cryptomining malware detection based on edge computing-oriented multi-modal features deep learning," *China Communications*, vol. 19, no. 2, pp. 174-185, 2022.
[35] A. Bensaoud and J. Kalita, "Deep multi-task learning for malware image classification," *Journal of Information Security and Applications*, vol. 64, p. 103057, 2022.
[36] A. Ikinci, T. Holz, and F. Freiling, "Monkey-spider: Detecting malicious websites with low-interaction honeyclients," in *SICHERHEIT 2008–Sicherheit, Schutz und Zuverlässigkeit. Beiträge der 4. Jahrestagung des Fachbereichs Sicherheit der Gesellschaft für Informatik eV (GI)*, 2008: Gesellschaft für Informatik e. V., pp. 407-421.
[37] Y. Choi, T. Kim, S. Choi, and C. Lee, "Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis," in *Future Generation Information Technology: First International Conference, FGIT 2009, Jeju Island, Korea, December 10-12, 2009. Proceedings 1*, 2009: Springer, pp. 160-172.
[38] A. Gorji and M. Abadi, "Detecting obfuscated JavaScript malware using sequences of internal function calls," in *Proceedings of the 2014 ACM Southeast Conference*, 2014, pp. 1-6.
[39] J. Wang, Y. Xue, Y. Liu, and T. H. Tan, "Jsdc: A hybrid approach for javascript malware detection and classification," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015, pp. 109-120.
[40] K. Kim *et al.*, "J-force: Forced execution on javascript," in *Proceedings of the 26th international conference on World Wide Web*, 2017, pp. 897-906.
[41] X. Hu, Y. Cheng, Y. Duan, A. Henderson, and H. Yin, "Jsforce: A forced execution engine for malicious javascript detection," in *Security and Privacy in Communication Networks: 13th International Conference, SecureComm 2017, Niagara Falls, ON, Canada, October 22–25, 2017, Proceedings 13*, 2018: Springer, pp. 704-720.
[42] S. Ndichu, S. Kim, and S. Ozawa, "Deobfuscation, unpacking, and decoding of obfuscated malicious JavaScript for machine learning models detection performance improvement," *CAAI Transactions on Intelligence Technology*, vol. 5, no. 3, pp. 184-192, 2020.

[43] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the association for computational linguistics*, vol. 5, pp. 135-146, 2017.
[44] A. Joulin, E. Grave, P. Bojanowski, and T. Mikolov, "Bag of tricks for efficient text classification," *arXiv preprint arXiv:1607.01759*, 2016.
[45] M. F. Rozi, S. Kim, and S. Ozawa, "Deep neural networks for malicious JavaScript detection using bytecode sequences," in *2020 International Joint Conference on Neural Networks (IJCNN)*, 2020: IEEE, pp. 1-8.
[46] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "{ZOZZLE}: Fast and precise {In-Browser}{JavaScript} malware detection," in *20th USENIX Security Symposium (USENIX Security 11)*, 2011.
[47] A. Fass, R. P. Krawczyk, M. Backes, and B. Stock, "Jast: Fully syntactic detection of malicious (obfuscated) javascript," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 15th International Conference, DIMVA 2018, Saclay, France, June 28–29, 2018, Proceedings 15*, 2018: Springer, pp. 303-325.
[48] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*, 2014: PMLR, pp. 1188-1196.
[49] S. Ndichu, S. Kim, S. Ozawa, T. Misu, and K. Makishima, "A machine learning approach to detection of JavaScript-based attacks using AST features and paragraph vectors," *Applied Soft Computing*, vol. 84, p. 105721, 2019.
[50] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.
[51] Y. Fang, C. Huang, Y. Su, and Y. Qiu, "Detecting malicious JavaScript code based on semantic analysis," *Computers & Security*, vol. 93, p. 101764, 2020.
[52] M. F. Rozi, T. Ban, S. Ozawa, S. Kim, T. Takahashi, and D. Inoue, "JStrack: Enriching malicious JavaScript detection based on AST graph analysis and attention mechanism," in *Neural Information Processing: 28th International Conference, ICONIP 2021, Sanur, Bali, Indonesia, December 8–12, 2021, Proceedings, Part II 28*, 2021: Springer, pp. 669-680.
[53] Y. Kim, "Convolutional neural networks for sentence classification proceedings of the 2014 conference on empirical methods in natural language processing, emnlp 2014, october 25-29, 2014, doha, qatar, a meeting of sigdat, a special interest group of the acl," *Association for Computational Linguistics, Doha, Qatar*, 2014.
[54] B. N. Narayanan, O. Djaneye-Boundjou, and T. M. Kebede, "Performance analysis of machine learning and pattern recognition algorithms for malware classification," in *2016 IEEE national aerospace and electronics conference (NAECON) and ohio innovation summit (OIS)*, 2016: IEEE, pp. 338-342.
[55] Y. Huang, T. Li, L. Zhang, B. Li, and X. Liu, "JSContana: Malicious JavaScript detection using adaptable context analysis and key feature extraction," *Computers & Security*, vol. 104, p. 102218, 2021.
[56] K. Cho *et al.*, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.
[57] N. M. Phung and M. Mimura, "Detection of malicious javascript on an imbalanced dataset," *Internet of Things*, vol. 13, p. 100357, 2021.
[58] J. W. Stokes, R. Agrawal, G. McDonald, and M. Hausknecht, "Scriptnet: Neural static analysis for malicious javascript detection," in *MILCOM 2019-2019 IEEE Military Communications Conference (MILCOM)*, 2019: IEEE, pp. 1-8.
[59] D. R. Patil and J. Patil, "Detection of malicious javascript code in web pages," *Indian Journal of Science and Technology*, vol. 10, no. 19, pp. 1-12, 2017.

[60] H. Liang, Y. Yang, L. Sun, and L. Jiang, "Jsac: A novel framework to detect malicious javascript via cnns over ast and cfg," in *2019 International Joint Conference on Neural Networks (IJCNN)*, 2019: IEEE, pp. 1-8.
[61] R. Banerjee, A. Baksi, N. Singh, and S. K. Bishnu, "Detection of XSS in web applications using Machine Learning Classifiers," in *2020 4th international conference on electronics, materials engineering & nano-technology (IEMENTech)*, 2020: IEEE, pp. 1-5.
[62] H. Li, S. Zhou, W. Yuan, J. Li, and H. Leung, "Adversarial-example attacks toward android malware detection system," *IEEE Systems Journal*, vol. 14, no. 1, pp. 653-656, 2019.
[63] Y.-M. Chen, C.-H. Yang, and G.-C. Chen, "Using generative adversarial networks for data augmentation in android malware detection," in *2021 IEEE conference on dependable and secure computing (DSC)*, 2021: IEEE, pp. 1-8.
[64] X. Peng, H. Xian, Q. Lu, and X. Lu, "Semantics aware adversarial malware examples generation for black-box attacks," *Applied Soft Computing*, vol. 109, p. 107506, 2021.
[65] Y. Ding, M. Shao, C. Nie, and K. Fu, "An efficient method for generating adversarial malware samples," *Electronics*, vol. 11, no. 1, p. 154, 2022.
[66] Z. Wang, K. W. Fok, and V. L. Thing, "Machine learning for encrypted malicious traffic detection: Approaches, datasets and comparative study," *Computers & Security*, vol. 113, p. 102542, 2022.
[67] B. Yuan, J. Wang, D. Liu, W. Guo, P. Wu, and X. Bao, "Byte-level malware classification based on markov images and deep learning," *Computers & Security*, vol. 92, p. 101740, 2020.
[68] T. M. Mohammed, L. Nataraj, S. Chikkagoudar, S. Chandrasekaran, and B. Manjunath, "Malware detection using frequency domain-based image visualization and deep learning," *arXiv preprint arXiv:2101.10578*, 2021.
[69] S. A. Khayam, "The discrete cosine transform (DCT): theory and application," *Michigan State University*, vol. 114, no. 1, p. 31, 2003.
[70] S. Euh, H. Lee, D. Kim, and D. Hwang, "Comparative analysis of low-dimensional features and tree-based ensembles for malware detection systems," *IEEE Access*, vol. 8, pp. 76796-76808, 2020.
[71] S. Ni, Q. Qian, and R. Zhang, "Malware identification using visualization images and deep learning," *Computers & Security*, vol. 77, pp. 871-885, 2018.
[72] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, 2002, pp. 380-388.
[73] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware images: visualization and automatic classification," in *Proceedings of the 8th international symposium on visualization for cyber security*, 2011, pp. 1-7.
[74] B. N. Narayanan and V. S. P. Davuluru, "Ensemble malware classification system using deep neural networks," *Electronics*, vol. 9, no. 5, p. 721, 2020.
[75] M. Alazab, M. Alazab, A. Shalaginov, A. Mesleh, and A. Awajan, "Intelligent mobile malware detection using permission requests and API calls," *Future Generation Computer Systems*, vol. 107, pp. 509-521, 2020.
[76] R. Taheri, M. Ghahramani, R. Javidan, M. Shojafar, Z. Pooranian, and M. Conti, "Similarity-based Android malware detection using Hamming distance of static binary features," *Future Generation Computer Systems*, vol. 105, pp. 230-247, 2020.
[77] K. Kong, Z. Zhang, Z.-Y. Yang, and Z. Zhang, "FCSCNN: Feature centralized Siamese CNN-based android malware identification," *Computers & Security*, vol. 112, p. 102514, 2022.
[78] L. N. Vu and S. Jung, "AdMat: A CNN-on-matrix approach to Android malware detection and classification," *IEEE Access*, vol. 9, pp. 39680-39694, 2021.

[79] Y. Ding, X. Zhang, J. Hu, and W. Xu, "Android malware detection method based on bytecode image," *Journal of Ambient Intelligence and Humanized Computing*, vol. 14, no. 5, pp. 6401-6410, 2023.
[80] K. Shaukat, S. Luo, and V. Varadharajan, "A novel deep learning-based approach for malware detection," *Engineering Applications of Artificial Intelligence*, vol. 122, p. 106030, 2023.
[81] X. Li, K. Kong, S. Xu, P. Qin, and D. He, "Feature selection-based android malware adversarial sample generation and detection method," *IET information security*, vol. 15, no. 6, pp. 401-416, 2021.
[82] S. Chen *et al.*, "Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach," *computers & security*, vol. 73, pp. 326-344, 2018.
[83] X. Chen *et al.*, "Android HIV: A study of repackaging malware for evading machine-learning detection," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 987-1001, 2019.
[84] A. Kamath, V. Bhatu, T. Paranjape, and R. Sawant, "Malware classification and defence against adversarial attacks," in *Proceedings of Data Analytics and Management: ICDAM 2021, Volume 2*: Springer, 2021, pp. 267-274.
[85] H. Rathore, A. Samavedhi, S. K. Sahay, and M. Sewak, "Are malware detection models adversarial robust against evasion attack?," in *IEEE INFOCOm 2022-IEEE conference on computer communications workshops (INFOCOm WKSHPS)*, 2022: IEEE, pp. 1-2.
[86] K. Shaukat, S. Luo, and V. Varadharajan, "A novel method for improving the robustness of deep learning-based malware detectors against adversarial attacks," *Engineering Applications of Artificial Intelligence*, vol. 116, p. 105461, 2022.
[87] K. Kong, L. Wang, Z. Zhang, Y. Li, D. Zhao, and J. Huang, "KFFPDet: Android malicious application detection system with assisted detection of adversarial samples," *Expert Systems with Applications*, vol. 252, p. 124095, 2024.
[88] S. Ndichu, S. Kim, S. Ozawa, T. Ban, T. Takahashi, and D. Inoue, "Detecting Web-Based Attacks with SHAP and Tree Ensemble Machine Learning Methods," *Applied Sciences*, vol. 12, no. 1, p. 60, 2021.
[89] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control-flow graph matching," in *Detection of Intrusions and Malware & Vulnerability Assessment: Third International Conference, DIMVA 2006, Berlin, Germany, July 13-14, 2006. Proceedings 3*, 2006: Springer, pp. 129-143.
[90] P. Pereira, J. Gonçalves, J. Vitorino, E. Maia, and I. Praça, "Enhancing JavaScript Malware Detection through Weighted Behavioral DFAs," arXiv preprint arXiv:2505.21406, 2025.
[91] Z. Liang, X. Wang, Z. Hu, L. Song, L. Chen, J. Guo, Y. Wang, and Y. Tian, "Breaking Obfuscation: Cluster-Aware Graph with LLM-Aided Recovery for Malicious JavaScript Detection," arXiv preprint arXiv:2507.22447, 2025.

7. Appendix A— Pseudocode of the Experimental Pipelines

This appendix presents the high-level pseudocode of the four experimental pipelines implemented in this study.

Algorithm 1 — Baseline Feature-Based Machine Learning Classification

Input:
Labeled Dataset D

Output:
Classification Metrics

1: Load Dataset D
2: Encode target labels
3: Separate features X and labels y

4: Initialize Stratified 5-Fold Cross Validation

5: FOR each fold DO

6: Split train/test data

7: Fit RobustScaler on training set

8: Transform training and testing sets

9: Train machine learning classifiers

10: Predict labels

11: Compute evaluation metrics:
AUROC
Accuracy
Precision
Recall
F1-score
Specificity
MCC
Cohen's Kappa
Macro / Weighted Metrics

12: END FOR

13: Report average results across folds
14: Plot ROC curves

Algorithm 2 — Graph-Based Node2Vec Representation Learning
Source file: Experiment2_Node2Vec_RepresentationLearning.ipynb
Input:
Labeled Dataset D

Output:
Graph-Based Classification Metrics

1: Load Dataset D

2: Encode labels

3: Separate X and y

4: Initialize Stratified 5-Fold Cross Validation

5: FOR each fold DO

6: Split train/test

7: Apply RobustScaler

8: Construct k-NN graph
k = 10

9: Convert graph to NetworkX

10: Train Node2Vec
dimensions = 64
walk_length = 20
num_walks = 3

11: Generate train embeddings

12: Generate test embeddings
using nearest neighbors

13: Train classifiers

14: Evaluate performance

15: END FOR

16: Compute average fold performance

17: Plot ROC curves

Algorithm 3 — CTGAN-Based Minority Class Augmentation
Source file: Experiment3_CTGAN_DataAugmentation.ipynb
Input:
Labeled Dataset D

Output:
Augmented Classification Metrics

1: Load Dataset

2: Encode labels

3: Separate X and y

4: Initialize Stratified 5-Fold Cross Validation

5: FOR each fold DO

6: Split train/test

7: Scale features

8: Build training dataframe

9: Identify minority class

10: Train CTGAN
only on minority samples

11: Generate synthetic samples

12: Assign generated labels

= minority class

13: Merge:
real + synthetic samples

14: Train classifiers

15: Evaluate on original test set

16: END FOR

17: Aggregate results

18: Plot ROC curves

Algorithm 4 — Hybrid Node2Vec + CTGAN Framework
Source file: Experiment4_Hybrid_Node2Vec_CTGAN.ipynb
Input:
Labeled Dataset D

Output:
Hybrid Classification Metrics

1: Load Dataset

2: Encode labels

3: Separate X and y

4: Initialize Stratified 5-Fold Cross Validation

5: FOR each fold DO

6: Split train/test

7: Scale training features

8: Construct k-NN graph

9: Train Node2Vec

10: Generate train embeddings

11: Generate test embeddings

12: Detect minority class

13: Train CTGAN
on minority embeddings

14: Generate synthetic embeddings

15: Merge:
original + synthetic embeddings

16: Train classifiers

17: Evaluate performance

18: END FOR

19: Compute average metrics

20: Plot ROC curves