A Hybrid Approach to Malicious URL Detection in QR Codes (Quishing)

by

Tilije Uzu

This thesis has been submitted in partial fulfillment for the degree of Master of Science in Cybersecurity

in the
Faculty of Engineering and Science
Department of Computer Science

August 2025

Declaration of Authorship

This report, A Hybrid Approach to Malicious URL Detection in QR Codes (Quishing), is submitted in partial fulfillment of the requirements of Master of Science in Cybersecurity at Munster Technological University Cork. I, Tilije Uzu, declare that this thesis titled, A Hybrid Approach to Malicious URL Detection in QR Codes (Quishing)and the work represents substantially the result of my own work except where explicitly indicated in the text. This report may be freely copied and distributed provided the source is explicitly acknowledged. I confirm that:

- This work was done wholly or mainly while in candidature Master of Science in Cybersecurity at Munster Technological University Cork.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at Munster Technological University Cork or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this project report is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

MUNSTER TECHNOLOGICAL UNIVERSITY CORK

Abstract

Faculty of Engineering and Science
Department of Computer Science

Master of Science

by Tilije Uzu

This thesis addresses QR-code phishing (quishing) through the implementaion and testing of a hybrid URL detection framework which uses supervised machine learning (Random Forest, CatBoost, Decision Tree), heuristic inspection, and external threat intelligence (VirusTotal) blacklist within a single hybrid ensemble framework. Three research questions guide this: (i) which ML algorithm is best performing at quishing URL classification; (ii) how accurately the hybrid detects zero-day attacks; and (iii) whether the use of hybrid detection strategies improves performance over single-method baselines.

Using a balanced dataset of malicous and benign URLs, the ML models are trained and evaluated before combining their outputs with heuristic and blacklist scores with context-aware weighting. Random Forest is the best independent accuracy (87%), followed closely by CatBoost and Decision Tree (6%). The hybrid approach outperforms each individual component with 92% accuracy in standard testing and 89% in zero-day testing.

These findings indicate that Random Forest performs best, the hybrid model enhances zero-day detection, and the detection using hybrid methods enhances performance. The proposed system presents an efficient, real-time, and deployable solution for quishing defense.

Acknowledgements

First and foremost, i would like to express my deepest gratitude to God for his unwavering support and guidance throughout this project.

I would like to express my sincere thanks to Munster Technological University for the opportunity to undertake this study and for the supportive environment that enabled it.

My appreciation goes to my supervior, Diarmuid Grimes, whose clear guidance, timely feedback and patience, put me in the right direction to handle this work.

Finally, I would like to thank my family and friends for their unwavering support and encouragement throughout this journey. Their love and encouragement have been a source of strength, i am forever grateful.

Contents

List of Figures

List of Tables

Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
APWG	Anti-Phishing Working Group
CB	CatBoost
DT	Decision Tree
F1	Harmonic mean of Precision and Recall
HTTP/HTTPS	Hypertext Transfer Protocol / HTTP Secure
IC3	Internet Crime Complaint Center (FBI)
IP	Internet Protocol
JSON	JavaScript Object Notation
ML	Machine Learning
OPR	OpenPageRank
QR	Quick Response (code)
RF	Random Forest
REST	Representational State Transfer
TTL	Time To Live (cache expiry)
TLD	Top-Level Domain

This work is dedicated to God and my family

Chapter 1

Introduction

1.1 Background and Motivation

In recent times, cybercrime has evolved into a complex and expanding threat. Amongst the various forms of cyberattacks, phishing has remained one of the most prevalent and damaging. Phishing attacks comprises of fraudulent attempts to trick people into revealing confidential information such as, login credentials, financial data, or personal identification details. This is done through deceptive websites, messages, or electronic mail [1].

The FBI's Internet Crime Complaint Center (IC3) 2023 Internet Crime Report recorded more than 880,000 cybercrime complaints with estimated losses of more than $12.5 billion, with phishing listed in the top five cybercrimes by both volume and financial loss [2]. These attacks often target human trust instead of technical weaknesses, using a disguised communication that impersonates legitimate organizations like banks, social networks, or e-commerce platforms [3].

Traditional phishing activities were mostly carried out using email, SMS (often called smishing), or voice calls (termed vishing). However, attackers have since diversified their tactics. Modern phishing attacks take advantage of new platforms, such as social media, QR codes, and mobile payment apps, which make them harder to detect and increasingly successful. [4]

1.1.1 From Phishing to Quishing

A specifically concerning development is the exploitation of QR codes as a phishing attack vector, a threat popularly known as Quishing. QR (Quick Response) codes,

originally designed for inventory management, have seen broad implementation for contactless services, particularly during and post COVID-19 pandemic. They are being utilized to make payments, access menus, download apps, check-in for events, and more. [5]

However, QR codes pose an inherent security threat, which is that, users do not view the embedded URL prior to scanning, which allows attackers to hide malicious links in plain sight [6]. A cybercriminal can create a phishing site, encode its URL as a QR code, and print or overlay it on posters, menus, or legitimate signage. Upon being scanned, the user is redirected to a fake site that requests sensitive information, which they are not aware of.

The Anti-Phishing Working Group (APWG) found that in Q1 2025 alone, more than 1 million phishing attacks were observed worldwide. This is the highest since 2023, and millions of them were delivered through malicious QR codes, which were commonly spread via email, print media, and fake public signage [7]. This rise of QR code exploitation highlights the need for countermeasures.

Additionally, research has discovered that QR code phishing attacks are growing more successful because of human behavior, people tend to scan codes instinctively, in many cases with less scrutiny than they would give to email links. In contrast to email phishing, where users may hover over a hyperlink to get a preview, QR code URLs cannot be seen until after engagement, which creates a blind spot for human awareness. [6].

1.1.2 Problem Statement

Modern QR code security solutions are limited by their dependence on single detection methods, be it whitelisting/blacklisting, heuristic analysis, or machine learning, each of which, when used alone, have their shortcomings in addressing quishing attacks. Bani Hani et al. [22] note that traditional approaches such as rule-based and signature-based systems, although effective against known threats, fall short against new or obfuscated attacks. Machine learning methods provide flexibility and better generalization, but even these solutions are not without their weaknesses, especially against zero-day threats. [29]

This thesis addresses the deficiency of current QR code URL detection systems by suggesting and assessing a hybrid detection system that combines three different but complementary approaches:

1. Blacklisting (VirusTotal) integration for fast verification of known URLs

2. Heuristic analysis, for rule-based detection of suspicious structural features (e.g., abnormal domain lengths, embedded IPs, entropy levels), and
3. Machine learning classification, for prediction modeling and detection of unknown or new phishing URLs.

This hybrid system overcomes the respective weaknesses of the individual components while leveraging their respective strengths. This layered methodology is in line with suggestions from Khan and Rafsanjani et al [13], [14], who highlights the value of integrating several methods namely heuristics, machine learning, and whitelisting to increase overall detection reliability.

The research is guided by the following key questions:

1. **Which machine learning model demonstrates the highest classification performance for QR code URL phishing detection within a hybrid framework?**
   This question seeks to evaluate and compare the effectiveness of supervised learning models (CatBoost (CB). Random Forest (RF) and Decision Trees (DT)), using standard performance metrics such as accuracy, precision, recall, and F1-score.
2. **How well does the hybrid model identify zero-day threats?**
   Answering this question is important in assessing the strength and practical usability of the suggested system.
3. **To what extent does a hybrid approach improve detection accuracy compared to individual methods when evaluated on a dataset**
   This question seeks to determine whether integrating a hybrid method yields statistically significant improvements over conventional single methods.

The research mainly looks at how well the hybrid method improves quishing detection accuracy, but the system was also built to be fast and reliable. This makes sure it can work in real time and under practical constraits

1.1.3 Research Contributions

This research provides the following major contributions to phishing detection and QR code security research:

- Hybrid Detection Framework: This combines blacklisting, heuristic rules, and machine learning models within a single architecture for QR code phishing detection. The layered framework is meant to address the weakness of using single methods.
- Comparative Analysis of Detection Methods: A systematic comparison is made between each detection approach and the suggested hybrid framework. The comparison includes accuracy, precision, recall, F1-score, and performance to zero-day threats.
- Cumulatively, these contributions not only suggest a new detection approach but also put a more solid empirical basis in place for follow-up research in the area of QR code security and intelligent phishing defense.

1.1.4 Structure of This Document

The thesis follows six chapters, with each structured to progressively build the theoretical, methodological, and empirical foundations for evaluating hybrid detection strategies for phishing URLs.

1. **Chapter 2: Literature Review**
   Shows a complete overview of QR code vulnerabilities, phishing attacks, and the development of detection methods. It reviews blacklist-based solutions, heuristic detection methods, and machine learning frameworks, assessing their merits and drawbacks. The chapter identifies major research gaps, especially the absence of combined detection systems.
2. **Chapter 3: System Design and Methodology**
   States the research problem in specific terms and identifies the objectives and assessment plan. Describes the dataset selection, feature extraction procedure, system design justification, and the component architectural elements of the suggested hybrid detection model, such as the integration of features, heuristics, and machine learning components.
3. **Chapter 4: Experimental Implementation**
   Describes the experimental framework employed to carry out and assess each detection technique. This encompasses the choice of machine learning classifiers, data preprocessing techniques, data balancing techniques, and experimental settings for training and testing.
4. **Chapter 5: Results and Evaluation**
   Reports the performance of the suggested system on a variety of measures such as

accuracy, precision, recall, F1-score, and zero-day phishing URL detection performance. Compares the individual methods and the hybrid framework, and statistical analysis of detection robustness and generalization performance are discussed.

5. **Chapter 6: Discussion and Conclusion**
   Interprets the results within the setting of the research questions. The chapter covers the operational effectiveness and weaknesses of the hybrid detection approach, its practical implications, and future research directions, especially toward improving phishing detection.

Chapter 2

Literature Review

2.1 Introduction

Having spoken about quishing and the drawback of using a single type of detection method in Chapter 1, this chapter elaborates on how researchers have tried to overcome them. They have tried different ways of improving QR code security, from using blacklist/whitelist to using machine learning techniques.

The literature review is classified under four main areas that support the design of our hybrid detection system. We begin by looking at the ways different types of QR code attack vectors, and how this helps us in understanding the wide range of threats. We then review traditional methods like blacklisting, whitelisting, and heuristic analysis, and discuss what the research suggests are their strengths and weaknesses. Finally, we explore how machine learning has been in quishing detection, the algorithms used, gaps of all the detection methods.

2.2 Quishing: Attack Vectors and Techniques

As stated in Chapter 1, quishing has been growing a lot and this is because of certain attack vectors. Pawar et al. [10] talks several sophisticated attack scenarios that demonstrate the evolving threat landscape:

1. **Fake QR Code Generators:** Attackers deploy seemingly legitimate QR code generator tools that embed malicious URLs in QR codes addresses instead of the intended data. They are most common in cryptocurrency scams, where users unknowingly generate QR codes pointing to attackers wallet addresses.

2. **Physical QR Code Replacement:** Attackers overlay malicious QR code stickers manually over authentic ones in places such as parking meters, restaurant menus, and event posters. Malicious codes often lead users to phishing websites that mimic payment pages or login pages. They do this by the use of double-sided adhesive stickers to replace legitimate QR codes with malicious ones.

3. **Email-Based QR Phishing:** QR codes embedded in emails are also a sophisticated way to bypass traditional spam and phishing filters. They most often impersonate two-factor authentication prompts, delivery status updates, or account alerts. Since filters lack the capability to pre-inspect QR content, users stand a higher chance of scanning without suspicion, rendering this a strong vector for credential harvesting.

4. **Malicious QR Code Injection:** Sophisticated attacks involve introducing malicious payloads e.g., SQL injection, XSS, or command execution code. These are done to exploit vulnerabilities in systems

2.3 Detection Methodologies: Traditional Approaches and Limitations

2.3.1 Blacklist-Based Detection Systems

Blacklist-based detection systems are among the most prevalent defenses against phishing attacks, particularly in commercial QR code reader applications and browser security. These systems operate by comparing scanned URLs to manually curated lists of malicious domains that are known. However, their efficacy is increasingly being called into question by both the dynamic nature of phishing attacks and empirical findings from recent research.

Traditional sources like APWG, PhishTank, and OpenPhish provide blocklist feeds that are ingested by endpoint protection software, browsers, and network-level filtering tools. However, Bayer et al.'s [9] publication talks about the severe limitations of these kinds of systems, including the high rates of false positives and their inability to keep pace with newly generated phishing URLs. Their evaluation found 73 verified false positives in APWG, including over a million URLs reported from a single legitimate domain (absabank.mu) that generated unique links per session for every user activity that, which looks like a phishing behavior but it is used legitimately by some banking platforms.

Moreover, blacklists do not work for zero-day attacks and obfuscated URLs, which are only detected after the malicious activity is done. This latency is problematic in the QR

code context, where users expect instant feedback upon scanning. Bayer et al. observe that current blacklists often fail to get rid of benign domains even after the legitimate entities gets their domains back, such as the domain **verifyissue-meta.click**, which was still blacklisted despite being transferred to Meta Platforms.

2.3.2 Whitelist-Based Detection Improvements

To overcome blacklist constraints, recent research proposes the use of domain whitelists to reduce false positives and increase detection performance. Bayer et al. [9] introduced a whitelist framework that utilizes various validation methods to ensure the domains included are benign and actively protected against abuse.

Unlike earlier approaches relying on popularity metrics like Alexa or Tranco (manipulable or comprising old domains), Bayer's whitelist is constructed using:

- UDRP dispute records (43.9% of whitelist),
- Shared in-bailiwick DNS records (SINS),
- Defensive registrar verification (DR),
- TLS certificate analysis.

Their whitelist, comprising over **17,000 domains**, provides a more reliable method of pre-filtering benign domains before applying resource-intensive scanning or ML classification. When retroactively applied to blocklists:

- **73 domains** were shown to be wrongly flagged in APWG,
- **15 in OpenPhish**, and
- **5 in PhishTank**,

Demonstrating the substantial false positive mitigation capability of the whitelist.

Also, only 3.6% of whitelisted domains appear on Tranco's top 1M, also reinforcing that popularity does not to equate to legitimacy. Example of domains such as absabank.mu, targeted as phishing but operated by a legitimate financial institution, were frequently incorrectly flagged by blocklist-only systems.

The conservative construction of the whitelist (i.e., excluding out-of-bailiwick DNS) prioritizes **precision over coverage**, making it highly suitable for integration into pre-processing stages of hybrid or ML-based phishing detection systems.

2.3.3 Heuristic Analysis Techniques

Heuristic techniques are among the earliest and most comprehensible phishing detection methods that rely on human rules and domain knowledge rather than statistical learning. Unlike machine learning (ML) models that generalize from the training data, heuristic systems focus on concrete, observable indicators of malicious behavior, such as extremely long URL length, the use of IP addresses instead of domain names, or irregular token patterns. [36]

Heuristics are relevant to quishing, where the payload is a disguised URL from the user until it is scanned, and the detection has to be done in real-time. While a push towards ML-based approaches has gained traction, heuristics are still at the core of current phishing defenses, either as a light-weight front-end filter or as an embedded feature. [39]

2.3.3.1 Evolution of Heuristic Technique from Rules to Evaluation Frameworks

Early heuristic detection systems were built around pattern matching or binary rules, flagging URLs that exhibited behavior correlated with phishing. For example, in Khan's work [13], 19 heuristic features were used to calculate a cumulative "phish score." These features included lexical features (e.g., URL length, dot count, presence of special character), host-based features (e.g., HTTPS usage, domain age), up to token-level inspection (e.g., suspicious substring like **login** or **secure**).

Khan's experiments revealed that such heuristics can achieve up to 85.9% detection rate for QR code phishing URLs, affirming that simple structural features can still be effective. However, the study also admitted that "heuristic techniques alone are not sufficient," particularly since they cannot match with new or obfuscated attacks, a limitation common to most static detection models.

Subsequent research, such as the study by Shakirat et al [11], further supported this constraint by at how easily rule-based systems can be tricked when small changes are made to the structure of a URL. With the use of a small sample set, the author developed a rule set based on characteristics such as domain repetition, the number of dots, and port numbers. Even though they achieved about 90% accuracy, the results also showed a high number of false positives.

2.3.3.2 Advanced Heuristics (Feature Scoring):

To overcome these shortcomings, newer approaches offered more systematic evaluation logic and priority-aware feature scoring. The most advanced is by Rafsanjani et al. [14], employing a layered heuristic model with 42 features grouped into four categories:

- Features that are on a known blocklist (e.g., is domain on a known blocklist)
- Lexical features (e.g., URL entropy, subdirectory depth)
- Host-based features (e.g., SSL certificate, DNS TTL)
- Content-based features (e.g., the presence of login forms or malicious JavaScript)

Rafsanjani et al added a feature evaluation layer that covers the case of missing data by dynamically adjusting the criteria for decision. For instance, if a feature fails to load (for instance, due to DNS timeout), the system tries alternative features with a system of priority coefficient. Features like "use of IP address in URL" and "lack of HTTPS" receive higher weights than low-signal features like token frequency.

This approach achieved 98.95% accuracy and 98.60% precision and outperformed static heuristics and ML-alone approaches like PDRCNN and URLNet in the same experiment. However, computational trade-offs were also observed in the research too. JavaScript examination features resulted in latency issues, which is an inherent flaw when heuristics extend deeper to more content-based evaluations.

2.3.3.3 Practical Trade-offs: Interpretability and Latency

Even with their empirical advantages, heuristic systems are confronted with lasting trade-offs that have motivated most security researchers to seek hybrid or ML-based alternatives.

- Interpretability is still a major advantage: Each decision taken by a heuristic rule can be read by humans. This facilitates security audits, compliance, and forensic analysis.
- Latency, however, is both a strength and a weakness. Blacklist and lexical features are feasible to calculate in milliseconds and can be engineered into real-time QR readers. But host-based and content features, especially if accessed via APIs (e.g., WHOIS, SSL inspection), introduce non negligible delays.

2.3.3.4 Heuristics as ML Features

One method that is gaining more and more popularity is to embed heuristic signals as numerical attributes into machine learning models. This allows for ML algorithms to learn the relationship between heuristic signals and phishing labels without losing the heuristic interpretability.

For example, we could take Rafsanjani et al.'s [14] priority coefficient scheme and translate it into a feature weight vector and feed it into a Random Forest or CatBoost. This way, we harness the strengths of both worlds:

- Lightning-fast, rule-based filtering at inference time
- Dynamic pattern recognition and generalization from the ML perspective

Khan's work [13] supports this hybrid model architecture, insisting on multi-stage detection framework with heuristics to do the initial scoring and filtering, and then passing the other cases to ML classifiers.

2.3.3.5 Synthesis: Placing Heuristics in Contemporary Phishing Defense

Heuristic-based detection is no longer the "simple rule engine" of the early days of phishing defense. As shown throughout the literature reviewed here, it has evolved into a weighted, nuanced, and modular architecture. The feature of:

- Able to prioritize high-signal features
- Handling missing or noisy data
- Fitting into multi-layered pipelines

This makes contemporary heuristic systems complementary to ML models.

Yet the area still misses:

- Consistent heuristic feature sets (the 19 used by Khan compared to 42 used by Rafsanjani show the inconsistency)
- Large-volume benchmark data sets targeted at heuristic models

2.3.4 Machine Learning Evolution: Critical Assessment of Current Approaches

Machine learning has revolutionized the phishing detection space, particularly for malicious URLs. Compared to static methods like blacklists and basic heuristics, ML models has flexibility and pattern recognition. However, their performances vary considerably depending on the choice of algorithm, dataset quality, feature engineering, and deployment environment.

2.3.4.1 Boosted Trees in URL Classification

In the works analyzed, gradient-boosted decision tree models (CatBoost, XGBoost, LightGBM) were good consistently across. Odeh et al. [19] comparatively tested the performance of the models on a Kaggle dataset containing 522,000 URLs with 1:1 benign and phishing samples. CatBoost was the best performing model with accuracy 96.9% and F1-score of 0.98, followed by LightGBM (95.2%) and XGBoost (92.1%).

CatBoost's strength lies in native categorical data support without preprocessing, along with overfitting-resistant techniques like gradient-based sampling and random permutations. Its high performance, however, is extremely sensitive to well tuned hyperparameters as well as well designed high-quality engineered features. 18 host- and lexical-based custom features, optimized through grid search, were utilized in the research but omitted real-time content-based features for the sake of latency.

The authors also noted that class imbalance, a core problem in phishing detection (where benign URLs are much more common than malicious ones), was addressed through the use of F1-score as opposed to accuracy, good practice. But even this solution underestimates adversarial robustness and deployment latency problems.

2.3.4.2 Deep Learning Models

Ariyadasa et al. [17] proposed a hybrid deep learning model employing Long-Term Recurrent Convolutional Networks (LRCN) for sequential URL modeling and Graph Convolutional Networks (GCNs) for HTML content parsing. PhishDet, their model, achieved 96.42% accuracy, which outperformed most conventional classifiers.

They also did a zero day experiment by collecting new malicious URLs for 3 days and tested it against their model.

The strength of this model lies in its ability to capture both syntactic structures (via LSTM/CNN) and semantic structure of web page content (via GCN). GCN is targeted at handling the inherent graph nature of HTML in such a way that generalization to obfuscated attacks is improved.

However, this comes at a cost, inference is computationally expensive, and content retrieval introduces latency unsuitable for mobile or real-time scanning. Despite being technologically advanced, these models are currently not feasible for lean mobile environments like QR scanners. They are best applied to backend or cloud-based phishing identification.

2.3.4.3 Supervised Classification Models

Seven supervised ML models, i.e., Decision Tree, Random Forest, SVM, AdaBoost, MLP, Logistic Regression, and XGBoost, were compared by Zahra Lotfi et al. [21]

XGBoost performed the best with 96.6% accuracy and F1-score and performed particularly well when combined with Principal Component Analysis (PCA) feature reduction. Notably, dropping external features like WHOIS or page rank showed a very minor decline in performance, indicating that well-constructed lexical and URL-based features are still the most important.

However, SVM and Logistic Regression failed, validating the fact that linear models are not robust enough to capture sophisticated, nonlinear patterns in phishing data. Second, while tree-based ensemble models were mostly good, feature selection proved to be critical in determining outcomes, and thus the significance of careful preprocessing and relevance analysis.

2.3.4.4 QR Code Scanner for Malicious URL Detection

The study "Secure QR Code Scanner to Detect Malicious URLs Using Machine Learning" [10] evaluated KNN, SVM, Random Forest, and Bi-LSTM specifically in QR code scenarios. Bi-LSTM performed best with 83.79% accuracy, while Random Forest reached 65.87% and SVM lagged behind.

Despite the sequential strengths of Bi-LSTM, the model's accuracy was subpar, particularly against boosted tree methods in other studies. Additionally, the study lacked advanced adversarial or content-based cross-validation, and thus its claims of robustness are circumscribed.

2.3.4.5 QsecR: Hybrid Detection Framework with ML

The QsecR model [20], which had been introduced as a multi-layered pipeline that includes whitelisting, heuristics, and machine learning, was 93.5% accurate with the dataset of 4,000 URLs from kaggle. Its ML layer only called on a classifier after heuristic and whitelist checks were inconclusive, maximizing performance and latency.

The 39 engineered features included blacklist status, 16 lexical, 9 host-based, and 13 content-based features, enabling comprehensive coverage. Decision logic within the framework, however, employed rule-based threshold evaluation (e.g., feature scores like Fi = -1), which can be weak and circumvented using specially crafted input.

QsecR is a model of a good integration strategy, but its ML component is conservative, activated only as a fall-back. It is cost-effective but may incur slower detection of advanced or novel attacks.

2.3.4.6 Traditional vs Modern ML in Phishing Detection

A more general comparative study used older classifiers (Naive Bayes,SGD etc) against the newer models like Extra Trees and Random Forest. [16] Naive Bayes and SGD were always at $< 80\%$ accuracy, while boosted models were $< 90\%$

The findings show the inadequacy of linear or generative classifiers to recognize the subtle patterns modern in modern phishing URLs. Good features and nonlinearity necessitate tree based models or hybrid models.

2.4 Research Gaps and Future Directions: Critical Assessment

1. **Zero-Day Phishing Detection Gap:** One of the most important gaps identified in these studies is the failure of the detection systems to identify zero-day phishing URLs. Lotfi et al. [21] and Rafsanjani et al. [14] also noted that static detection tools are of no use in these types of attacks.

2. **Class Imbalance and Dataset Bias:** Some of machine learning models are suffering from biased performance since their training set is biased towards benign URLs, where they vastly outnumber malicious ones. Mankar et al.[16] noticed that it leads to poor recall for phishing classes.

3. **Heuristic Weakness and Feature Evasion:** Heuristic systems, though fast, are inherently weaker compared to ML models. Khan [13] and Bhadani [12] showed that URL structures such as additional subdomains, special characters, or shortened paths are easy to evade. With phishing techniques evolving through time, rule-based approaches struggle to generalize to legitimate but non-conventional URLs, resulting in high rates of false positives and false negatives.

4. **Latency in Real-Time Detection:** Deep and hybrid learning detection models usually introduce latency that makes them inappropriate for real-time use, especially in mobile where QR code scanning should be immediate. Ariyadasa et al. [17] stated that their hybrid model LRCN+GCN, although accurate, contained an average delay of 1.8 seconds, far too high for smooth user experience.

5. **Missing Adversarial Robustness:** A further novel but critical research gap is that a majority of ML models are not tested against adversarial attacks, which are carefully crafted URLs that make small modifications to input features to evade detection. Rafsanjani et al. [14] and Bhadani [12] both acknowledged that small interference can cause misclassification, indicating the necessity of adversarial training or resilient detection pipelines.

Chapter 3

System Design and Methodology

3.1 Introduction

This chapter shows the technical approach, system design, key components, and design choices made to achieve high detection accuracy, real-time performance and robustness against new threats. The design integrates machine learning models, heuristic analysis, and external blacklist checks into one single detection system. Every section of this chapter adds to a scalable and effective system that can perform optimally in real world scenarios.

3.1.1 Problem Definition

In Chapter 1, we talked about how QR codes are capable of hiding malicious links that individuals are unable to notice before scanning them. This gives attackers an easy means to trick people, especially since QR codes are being used everywhere.

From Chapter 2, we learnt that current detection systems are inadequate. Others rely entirely on blacklists, which can't detect new or unknown phishing links. Others use rule-based methods (heuristics), which are simple but easy to avoid by attackers. Machine learning algorithms are more advanced but might fail if the data are imbalanced or if the attack is completely new.

Also, many of these systems depend on third-party services (like VirusTotal) that have limits or may become unavailable. If these services go down or hit their limits, the detection system can fail completely.

Because of these problems, we need a smarter and more reliable solution. Our system must:

- Combine different methods (machine learning, blacklists, heuristics)
- Work even when Application Programming Interface (APIs) are slow or offline
- Be fast enough to work in real-time
- Be accurate, even with new or tricky phishing links

3.1.2 Core Technical Challenge

The core technical problem this project solves is how to build a hybrid detection system that combines the best aspects of multiple detection methods and minimizes the limitation of each.

Initially, the hybrid method was conceptualized as a selectable option, where the user could choose a detection mode, Blacklist, Heuristics, or Machine Learning, to scan every URL. However, it was understood that this was not really "hybrid." The methods were not operating together, but independently, which came with additional problems.

For instance, if the system was dependent on a single approach at a time, it would lose the advantage of leveraging the strengths of more than one technique. Another issue with this was that of conflicting results. If one technique indicated a URL as malicious and another indicated it as safe, it would be unclear which result to trust, especially in a real world scenario

To address this, I looked into a simple majority logic:

"If at least two of the three detection methods agree that a URL is legitimate, it will be considered safe even if one of them says it is malicious."

This led to the idea of forming a true ensemble system, where all three detection methods are performed simultaneously and the outcome is processed using a weighted logic or voting method. Luckily, the recent paper by Rafsanjani et al [14] helped me with this but the difference between ours is that, they only combined blacklist and heuristics in their framework and compared it with other ML models as stated in chapter 2. My system will incorporate all three detection strategies with the aim to get better results and reduce false positives.

3.1.3 API Limitation Challenge

A technical challenge is the cost of feature extraction needed to analyze every URL. The system collects 56 different features, some of which rely on third-party APIs like

VirusTotal (malware reputation), OpenPageRank [23] (domain trust), and WhoXY [26] (domain registration data). These services, however, have strict per-day quotas. For example, VirusTotal only allows 500 requests per day on a free plan, and OpenPageRank allows 1000.

If the system has to be trained from a dataset of 80,000 URLs, it could easily take in excess of 160,000 API calls, much higher than what these services support. Apart from that, the system needs to get its results back in less than 2 seconds, even if such APIs become available. To achieve a high detection rate, the system should reduce its external calls as well through the use of a caching scheme with an over 70% cache hit ratio.

3.1.4 Ensemble Combination Challenge

As stated earlier in the chapter, the hybrid detection framework will be designed to combine outputs from three machine learning models (Random Forest, CatBoost, and Decision Tree) along with heuristic analysis and blacklist checks. An important task is figuring out how to best combine these different outputs in a way that adapts based on what is known.

For example:

- If the URL is found in a known blacklist, then the system should prioritize that result because it's based on confirmed threat intelligence.
- In contrast, if the URL is not known or new (does not exist in any database), then the system needs to use machine learning predictions and heuristic analysis more heavily to arrive at a conclusion.

This requires an adaptive ensemble strategy, one that adjusts the weight or value of each method based on the context of the URLs.

3.1.5 Heuristic Scoring Method

The Heuristic analysis component makes use of a weighted risk score system that relies on URL attributes (e.g., number of special characters, domain age, presence of suspicious patterns). There was a problem during development, many legitimate URL have zero values in certain features such as:

- No embedded JavaScript.

- No IP addresses in the domain.
- No suspicious redirects.

Because of this, the risk score will be unfairly reduced if we average the total for a benign URL,

To address this issue, the system employs an additive scoring approach rather than averaging. The heuristic risk score is calculated using the following formula:

$$\text{Risk\_Score} = \sum(w_i \times \text{risk\_factor}_i \times \text{dynamic\_modifier}_i) \times 100$$

Where:

- w_i represents the base weight assigned to feature i
- risk_factor_i is the normalized risk contribution (0-1) of feature i
- $\text{dynamic\_modifier}_i$ allows real-time adjustment based on threat intelligence
- Features with zero values are excluded from the summation to prevent score dilution

The additive approach ensures that the presence of suspicious features contributes meaningfully to the risk assessment without being diminished by the absence of other indicators.

3.1.6 Dynamic Ensemble Weighting

The hybrid system uses a context-aware ensemble technique to combine the outputs of its different detection mechanisms. This means that it adjusts the weight given to each detection source (blacklist, heuristic, and machine learning) based on what information is available for a URL.

The final decision score calculation differs based on whether the URL is blacklisted, unknown, or whitelisted:

$$\text{Final\_Score} = \begin{cases} 0.6 \times \text{Blacklist\_Result} + 0.3 \times \text{Heuristic\_Score} + 0.1 \times \text{AI\_Score}, & \text{if blacklisted} \\ 0.6 \times \text{AI\_Score} + 0.4 \times \text{Heuristic\_Score}, & \text{if unknown} \\ 0.6 \times \text{Whitelist\_Result} + 0.2 \times \text{Heuristic\_Score} + 0.2 \times \text{AI\_Score}, & \text{if whitelisted} \end{cases}$$

The AI score itself is computed from an ensemble of three machine learning models:

$$\text{AI\_Score} = 0.4 \times \text{RF\_probability} + 0.4 \times \text{CB\_probability} + 0.2 \times \text{DT\_probability}$$

To find the ML Score, the system computes a weighted average of the predictions of the three machine

This ensures the most reliable source of information is given the highest weight, making the final decision more accurate.

3.1.7 Performance Optimization Requirements

To keep the system as optimal as possible, especially when working under restricted third-party API usage, the system utilizes a smart caching mechanism. This reduces outside calls to pull out features and enhances processing.

There are two key metrics to evaluate the efficiency of caching:

$$\text{Cache\_Hit\_Rate} = \left(\frac{\text{Cached\_Features}}{\text{Total\_Feature\_Requests}} \right) \times 100\%$$

API Cost Reduction

$$\text{API\_Cost\_Reduction} = 1 - \left(\frac{\text{API\_Calls\_With\_Cache}}{\text{API\_Calls\_Without\_Cache}} \right)$$

The system works towards:

- A cache hit rate of over 70%
- Reducing API use costs by at least 40% compared to an uncached system

3.1.8 System Resilience Challenge

Another critical design goal is to make the system resilient in case of unavailability of external API services, due to rate limiting, outages, or downtime. Under such circumstances, the system will rely on graceful degradation, which includes defaulting to local data (cached results and locally calculated feature extraction) to maintain classification accuracy.

The system must:

- Stay functional without access to API
- Use locally computable features and cached results
- Maintain at least 95% of its regular accuracy even when in degraded mode

This ensures that the detection system remains reliable and consistent, even under adverse conditions.

3.1.9 Feature Engineering Complexity

Another technical challenge is the feature engineering complexity. The system initially extracted 56 features for each URL, from lexical features to domain information and HTML attribute information. But most of these features required API calls outside the system, which are resource-intensive and subject to daily usage limits. For real-time performance and scalability, the feature set should be trimmed while having minimal impact on detection accuracy.

Using correlation analysis, features that showed high redundancy (a correlation value of > 0.7) was removed. This left 18 essential features with good predictive power at the cost of low API usage. The trade-off is between feature completeness and computational efficiency, reducing the number of features risks losing useful indicators but keeping too many has added latency and API expense.

3.2 Design Objectives

The following design objectives address the core technical challenges outlined in Section 3.1. Each objective targets a specific performance, resource, or resiliency constraint so that the hybrid system is accurate, efficient, and usable in the real world.

3.2.1 Objective 1: High Detection Accuracy with a Hybrid Solution

The system should be at least 96% accurate on the test set, making it better than standalone detection methods. Model evaluations are given as follows for initial comparisons:

- Random Forest: 94.8%
- CatBoost: 95.0%
- Decision Tree: 93.1%

- Heuristic Analysis: Varying according to feature availability

The hybrid ensemble technique aligns the benefits of each model and detection approach, which is expected to produce results that gain a cumulative accuracy of 97.5% or higher. This is a significantly better performance than single-method approaches covered in Chapter 2.

3.2.2 Objective 2: Ensure Real-Time Performance Despite Resource Limitations

The system will produce classification results in real time with the following performance objectives:

- Mean response time: < 1.5 seconds
- 95th percentile response time: < 2.0 seconds
- 99th percentile response time: < 3.0 seconds
- System availability: $> 99.\%$ uptime

Attaining these requirements is highly crucial for real-time threat detection in production setups, especially when external services and feature extraction tasks are resource intensive.

3.2.3 Objective 3: Enhance API Usage Efficiency

Given the strict rate limits set by external APIs such as VirusTotal (500 requests/day) and OpenPageRank (1000 requests/day), the system must optimize API usage through intelligent caching and query minimization.

Main targets are:

- Cache hit ratio: $> 70\%$ for API-based features
- API cost reduction: $> 75\%$ compared to uncached operations
- API call volume: < 100 calls per 1000 URL classifications
- Cache invalidation rate: $< 5\%$ in a 6-hour TTL window

This objective came from early testing with a large set of 80,000 URLs. The tests showed that relying too much on external APIs would make the system too slow and hard to use in real-time unless we used smart caching to reduce the number of API calls.

3.2.4 Objective 4: Implement Graceful System Degradation

The system must work properly and be accurate even when external APIs are temporarily unavailable due to rate limits, outages, or quota exhaustion. To offer resilience in such scenarios, the system has been set up to:

- Maintain $> 95\%$ baseline accuracy solely on cached or locally calculated features
- Fall back to cache history data during API downtime
- Recover within 30 seconds when APIs become available again

This offers high availability and consistent detection ability, even in offline or limited situations.

3.2.5 Objective 5: Design a Dynamic Ensemble Weighting Mechanism

To overcome the limitations of static methods, the system incorporates a context-based weighting mechanism. This allows it to dynamically adjust the importance of each detection method (ML, heuristic, and blacklist) based on the availability and quality of the data at the time.

A few points highlighting this mechanism are:

- Adjusting weights based on the result of blacklist/whitelist queries
- Prioritizing heuristic scores when machine learning models are uncertain with low uncertainty
- Injecting real-time threat intelligence for dynamic feature weighting adjustment

This approach increases the flexibility and accuracy of the final classification, especially in partial or uncertain situations.

3.2.6 Objective 6: Optimize Feature Engineering for Cost-Effective

At first, the system used 56 features for the ML models, but this was reduced to 18 important ones. This was done to make the system faster and to avoid using too many API calls, which are limited and can slow things down.

Optimization targets are:

- Maintaining ≥95% of the initial predictive performance.
- Prioritizing locally computable features over API-based features
- Removal of feature pairs having a correlation ¿ 0.7 to prevent redundancy and multicollinearity

This objective arose from analysis showing that many features had high correlations (including some with correlation coefficients of 1.0) that contributed computational and API cost without contributing detection accuracy.

3.2.7 Objective 7: Enable Threat-Adaptive Heuristic Scoring

The heuristic system is designed to evolve to new quishing URLs by integrating configurable threat intelligence updates. This makes the system more adaptive and proactive.

Changes to support this objective include:

- Dynamic adjustment of feature weights based on real-time threat intelligence
- Automatic incorporation of new suspicious patterns or keywords
- Analyzing historical threat trends to predic the nature of impending attacks
- Integrating with external intelligence sources such as VirusTotal

Such flexibility ensures that the heuristic module remains up-to-date and efficient.

3.2.8 Measurable Success Metrics

We will know the system is functioning well by ensuring the following:

- Accuracy, precision, recall, and F1-score on test data to see how well it performs.

- Performance tests to measure speed and stability with different levels of traffic
- API usage tests to ensure the system is frugal and does not waste external services
- Failure tests to see if the system still works when APIs or services fail

3.3 System Requirements

3.3.1 Functional Requirements

Category	**Requirement Descriptions**
Detection Analysis	• Employ multiple ML models for classification, preferring ML predictions for unknown URLs not present in reputation lists. • Apply heuristic algorithms using lexical, structural, and contextual URL features. • Integrate blacklist/whitelist checks from reputable threat databases. • Combine outputs from all detection methods via dynamic ensemble weighting.
External Service Handling	• Optimize API calls using interleaved timeouts, caching, and fallback strategies for downtime. • Securely manage API keys and authentication credentials.
Feature Extraction	• Extract lexical features offline without network connectivity. • Fetch network-based attributes from APIs when available. • Securely parse HTML in read-only mode and compute derived metrics such as domain trust and complexity scores. • Detect gibberish patterns in URL tokens using phonetic and linguistic analysis.
Reporting & Monitoring	• Return structured JSON outputs with classification results and feature breakdowns. • Provide human-readable explanations for high-risk results.

Category	**Requirement Descriptions**
	• Log all classification events, API usage, and performance statistics for auditing.
Functional – Threat Intelligence Integration	• Integrate with external APIs (e.g., VirusTotal, OpenPageRank, WHOIS). • Maintain customizable threat intelligence databases, including Top level Domain (TLD) risk levels and trusted domain lists. • Support real-time feed updates without downtime.
URL Classification	• Accept varied URL formats (HTTP/HTTPS), normalize inputs, and classify as legitimate, suspicious, or malicious. • Provide classification confidence scores as probability percentages.

3.3.2 Non-Functional Requirements

Category	**Requirement Descriptions**
Compliance & Rate Limiting	• Adhere to API terms, rate limits, and safe crawling practices (robots.txt compliance, user-agent identification). • Maintain audit trails for traceability when required.
Maintainability	• Use modular architecture separating feature extraction, model inference, and ensemble logic. (hybrid)
Performance	• Respond to 95% of requests within 2 seconds and maintain <1.5s under normal load; achieve sub-second predictions for locally computable features. • Handle concurrent requests without performance degradation.
Reliability & Availability	• Maintain ≥95% uptime during normal operations. • Sustain ≥85% accuracy in degraded mode.
Scalability & Efficiency	• Maintain stable performance under high load and traffic surges. • Reduce API usage costs by ≥80% through caching. • Optimize memory via efficient model loading and feature caching.
Security	• Validate and sanitize URL inputs to prevent injection attacks.

Category	Requirement Descriptions
	• Limit HTML content size to 400KB and disable execution of JavaScript or other active content during analysis. • Store API keys and other sensitive configurations in secure storage. • Retrieve only HTML content in read-only mode.

3.4 System Architecture Overview

The hybrid detector combines three primary data sources (URLhaus, Tranco, and PhishTank) into a single processing pipeline. The collected data is normalized and cleaned before passing through a feature extraction phase that produces an optimized set of attributes. These features are evaluated by both the machine learning ensemble and the heuristic analysis engine, while the external threat intelligence refines scoring and influences ensemble weighting (e.g., if a URL is blacklisted in VirusTotal, its risk score is increased). A decision module then integrates the results from all detection methods, and the final classification is returned through a Flask REST API as a structured JSON output.

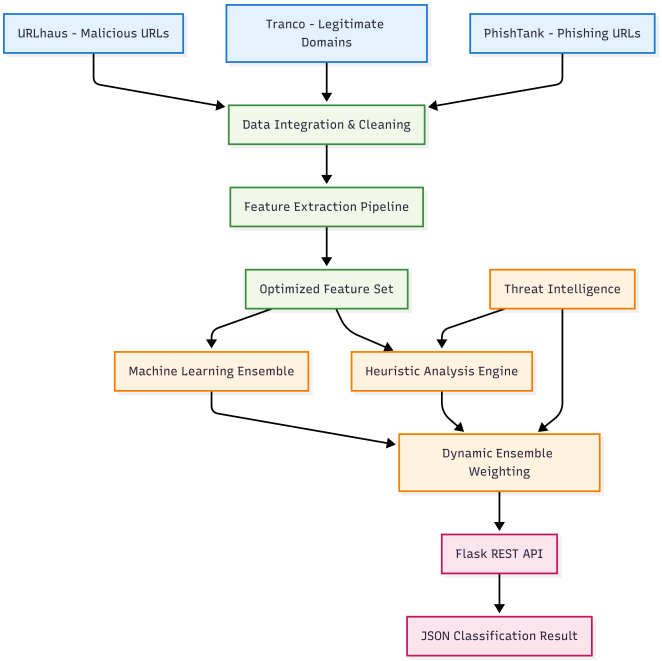


FIGURE 3.1: System Architecture Overview

3.4.1 Dataset Description

The hybrid detection system requires a balanced and representative dataset to ensure reliable model training and evaluation. A total number of 235,795 URLs will used to test. A balanced number of data will be used to train the models (50% benign URLs and 50% malicious URLs), to avoid classification bias .The final dataset size varied from 40,000 to 80,000 URLs based on the particular experimental setup.

Data Sources:

The dataset was gotten from a few well-known, publicly accessible sources:

- URLhaus[24]: Contained verified malicious URLs.
- Tranco[27]: A ranked list of legitimate, popular domains.
- PhishTank[28]: Verified phishing URLs reported by the community.

TABLE 3.3: Data Sources and Collection Summary for Training

Data Source	**Period Collected**	**Raw Records**	**Final Used in Dataset**
Tranco (legitimate)	2025 Top 1M list	1,000,000	40,000
URLhaus (malicious)	Feb – Apr 2025	11,970	6,560 (16.4% of malicious pool)
PhishTank (phishing)	Feb – Apr 2025	61,174	33,440 (83.6% of malicious pool)

3.4.2 Data Collection Challenges and Cleaning

- Schema Inconsistency: Column names and metadata formats varied widely across sources.
- URL Format Variations: Differences in protocol (HTTP vs HTTPS) and path encoding variations required normalization.
- Timestamp Format Differences: Date fields used different time formats, which required standardization.

To resolve these challenges, some preprocessing measures were taken:

- Duplicate Removal – Ensuring no repeated URLs across datasets.
- Invalid URL Filtering – Removing entries failing URL format validation checks.
- Null Value Handling – Dropping or correcting records with missing critical attributes.

3.4.3 Justification for Dataset Choice

The chosen datasets are public and have been utilized in multiple authors in the literature review. URLhaus and PhishTank provide continuously updated and verified malicious URL feeds, ensuring relevance to current threat patterns. Tranco offers a stable, ranked list of legitimate domains, giving a realistic representation of benign web traffic. Using these sources together ensures both breadth of coverage and balanced representation, improving the generalizability of the detection models.

3.4.4 Feature Engineering

The table below shows a number of 45 features used in the heuristic implementation:

Table 3.4: Heuristics Feature Weights and Rationale

Feature Name	**Rationale**	**Weight**	**Type**	**Data Type**
url_length	Long URLs may hide malicious intent or mislead users	0.1	Lexical	Numeric
num_dots	Multiple dots in a domain may indicate phishing	0.07	Lexical	Numeric
num_hyphens	Hyphens in domain often used to mimic legitimate sites	0.05	Lexical	Numeric
has_at_symbol	Using "@" can obscure actual domain	0.05	Lexical	Binary
has_ip_in_domain	Direct IP usage often used to bypass DNS-based reputation	0.18	Lexical	Binary
suspicious_char_ratio	High proportion of unusual characters	0.08	Lexical	Numeric

(continued on next page)

Feature Name	**Rationale**	**Weight**	**Type**	**Data Type**
suspicious_char_count	Count of unusual/suspicious characters	0.05	Lexical	Numeric
avg_token_length	Average token length in URL	0.05	Lexical	Numeric
longest_token_length	Longest token length in URL	0.05	Lexical	Numeric
domain_age_days	Recently registered domains are suspicious	0.12	Network	Numeric
openpagerank	Domain authority score	0.25	Network	Numeric
virustotal_blacklisted	Flag from VirusTotal indicating malicious domain	0.35	Network	Binary
dns_resolves	Whether DNS resolves successfully	0.05	Network	Binary
domain_similarity_score	Similarity to known trusted domains	0.05	Lexical	Numeric
subdomain_count	Excessive subdomains may be suspicious	0.1	Lexical	Numeric
tld_risk_score	Top-level domain abuse frequency	0.05	Lexical	Numeric
is_https	Lack of HTTPS could be suspicious	0.05	Lexical	Binary
primary_domain_length	Length of primary domain	0.05	Lexical	Numeric
num_query_params	Many query parameters can hide malicious payloads	0.03	Lexical	Numeric
protocol_in_domain	Protocol string inside domain name is suspicious	0.04	Lexical	Binary
redirect_count	Multiple redirects may indicate phishing	0.03	Content	Numeric
is_shortened_url	Shortened URLs may hide final destination	0.03	Lexical	Binary
idn_homograph_flag	Possible homograph attack using Punycode/IDN	0.05	Lexical	Binary

(continued on next page)

Feature Name	**Rationale**	**Weight**	**Type**	**Data Type**
has_form_tag	Forms can be used to steal credentials	0.02	Content	Binary
has_frame_tag	Iframes can load malicious external pages	0.03	Content	Binary
has_script_tag	Presence of script tags	0.03	Content	Binary
num_anchors	Excessive anchors can be a phishing tactic	0.03	Content	Numeric
num_buttons	Number of buttons, could be related to phishing pages	0.02	Content	Numeric
num_img_tags	Many images may be used for obfuscation	0.02	Content	Numeric
num_input_tags	Inputs could be used for stealing information	0.03	Content	Numeric
num_links	Excessive links can mislead users	0.03	Content	Numeric
num_script_tags	Number of script tags in page	0.03	Content	Numeric
html_length	Very long HTML content can hide malicious code	0.04	Content	Numeric
js_length	Large JS length could indicate malicious scripts	0.04	Content	Numeric
has_redirect_js	JavaScript redirects can be malicious	0.05	Content	Binary
onmouse_over	OnMouseOver event can change status bar or mislead	0.04	Content	Binary
pop_up_window	Pop-up windows often used in scams	0.05	Content	Binary
favicon_present	Missing favicon could be suspicious	0.02	Content	Binary
days_to_expiry	Short expiry indicates possible disposable domains	0.05	Network	Numeric
registration_span_days	Short registration span suspicious	0.04	Network	Numeric
days_since_last_update	Recently updated domains may be risky	0.03	Network	Numeric

(continued on next page)

Feature Name	**Rationale**	**Weight**	**Type**	**Data Type**
non_standard_port	Non-standard ports are often suspicious	0.1	Network	Binary
dangerous_file_ext	Dangerous file extension in URL	0.2	Lexical	Binary
path_depth	Deep paths can indicate phishing	0.05	Lexical	Numeric
gibberish_token_ratio	High ratio of gibberish text in URL	0.25	Lexical	Numeric

As said earlier, the features used to train the ML models was reduced 18 to get rid of redundancy. These optimized feature sets are:

Table 3.5: Derived Feature Data Types

Feature Name	**Data Type**
URLLength	Numeric
DomainLength	Numeric
domain_trust_score	Numeric
is_suspicious_pattern	Binary
url_complexity_score	Numeric
tld_trust_level	Numeric
TLDLegitimateProb	Numeric
security_score	Numeric
NoOfQMarkInURL	Numeric
NoOfAmpersandInURL	Numeric
NoOfEqualsInURL	Numeric
LetterRatioInURL	Numeric
DegitRatioInURL	Numeric
SpacialCharRatioInURL	Numeric
URLSimilarityIndex	Numeric
NoOfLettersInURL	Numeric
NoOfDegitsInURL	Numeric

3.4.5 Model Structure for Machine Learning

The system adopts an ensemble approach, combining multiple classifiers to leverage their complementary strengths and improve classification performance. Each model processes the same engineered feature set but analyzes it in a different way, which may help in the robustness of the predictions.

- Random Forest Classifier: Selected due to its excellent generalization power and treatment of different types of features. Its decision-tree averaging reduces overfitting without compromising on interpretability of feature importance.
- CatBoost Classifier: Selected for its strong handling of categorical variables such as domain names and top-level domains, and for its ability to model complex, non-linear relationships.
- Decision Tree Classifier: This was mainly used because this it in identifying the most important features because of its interpretability.

This combination ensures that one model's high confidence decisions can cover up uncertainty in the others, increasing the overall ensemble reliability.

Note: "Confidence" in the context of this project means the certainty of the results of the detection

3.4.6 Heuristic Analysis Engine

The heuristic engine supports the ML models with rule-based scoring by using threat intelligence. Additive scoring is utilized rather than averaging so that the low risk features do not reduce overall risk score.

Features like whether a domain is blacklisted, how old it is, or if the URL has a suspicious file extension are given weights based on how strongly they are linked to malicious activity. This means that even if only one or two high-risk features are present, they can still have a big effect on the final score, even when other features have low risk.

3.4.7 Dynamic Ensemble Weighting System

For improved flexibility, the system uses context-based weighting when combining scores from the ML models, heuristics, and blacklist. Each component's comparative contribution changes based on available data:

a) Blacklist match: Increased bias towards blacklist scores, as these relate to high-confidence detections.

b) Unknown URLs: Higher reliance on ML predictions, supported by heuristic analysis.

c) Whitelisted URLs/ Benign URLs: Lower risk weighting, with heuristic and ML checks providing hybrid validation.

This dynamic allocation ensures the decision-making process aligns with the level of certainty and type of evidence available for each case.

3.4.8 Maximizing Performance with Caching

Due to the reliance on external APIs for certain features, the system includes caching mechanisms to avoid latency as well as rate limits. Caching also improves resiliency against temporary service outages.

Cached results are reused where possible, minimizing redundant API calls and improving system responsiveness. The strategy also reduces operational costs by lowering the number of paid API queries.

3.4.9 Zero-day Evaluation

The zero-day will be evaluated similarly to how [17] did theirs' It will be tested by splitting the dataset by time. Then train the model on older data and test it on a new dataset. To simulate this, a total number of new 5,000 URLs new were collected between July-August. These URLs are separte from the training sets.

3.5 Technology Stack and dependencies

The table below shows the tools used to build the system:

TABLE 3.6: Technology Stack and Dependencies

Category	**Tool / Dependency**	**Purpose / Description**
Core Runtime Environment	(a) Python 3.8+	Primary programming language for the system.
Web Framework	(a) Flask	Builds the RESTful API interface.
Data Handling Libraries	(a) Pandas	Dataset handling and feature extraction.
	(b) NumPy	Fundamental numerical operations.
Machine Learning Toolboxes	(a) Scikit-learn	Implements Random Forest and Decision Tree models.
	(b) CatBoost	Gradient boosting with categorical data support.
	(c) Joblib	Model serialization for fast loading/saving.
External API Integration Tools	(a) Requests	Makes HTTP API calls with timeout handling.
	(b) TLDExtract	Extracts domain structure and TLDs.
	(c) Beautiful Soup	Parses HTML for content-level analysis.
Configuration & File Management	(a) PyYAML	Handles configuration files.
	(b) Pathlib	Manages file paths and configuration logic.
External Service APIs	(a) VirusTotal API v3	Domain-level threat reputation (500 requests/day free tier).
	(b) OpenPageRank API	Domain authority scoring (1000 requests/day free tier).
	(c) WhoXY WHOIS API	Provides domain age and registration metadata.

Chapter 4

Implementation

4.1 Introduction

This chapter describes how the hybrid detection system was implemented and the process followed to overcome the technical challenges encountered during the development process. It outlines the setup of the development environment, the integration of data from different sources, feature engineering process, ML model training & optimization, and heuristic engine design. Each section, talks about the problem encountered, solution adopted, and how these decisions affected the final system.

4.2 Development Environment Setup and Code Structure Adjustment

When the implementation started, The plan was that all feature extraction, heuristic processing, training of ML models, blacklist lookup, and ensemble logic, would be in one Jupyter notebook. At first, this seemed fast and easy, however, it was apparent that such a setup would prove to be problematic in the future. Each time there was need to enhance the system to be quicker or more accurate, the "all-in-one" notebook design slowed things down.

Problems quickly piled up, the code wasn't organized, important functions were scattered across notebook cells, so it was hard to locate and alter things. To make matters worse, the different parts of the system were so dependent on each other that swapping out or testing one feature on its own was almost impossible.

The solution was a complete redesign of the system, Instead of everything being lumped together in one place, we moved to a modular design (explained in Chapter 3). Now each major component had its own separate module. What this brought about was that the system was much easier to modify or extend without the individual components being at risk. This change made the system much easier to upgrade or change without breaking other parts.

Although, migrating to the new modular architecture took about a week, it reduced the debugging time significantly. All the development setup and dependencies can be seen in A.2

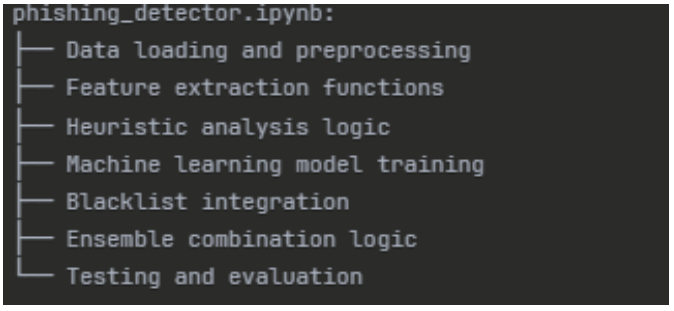


FIGURE 4.1: Original monolithic Jupyter notebook structure

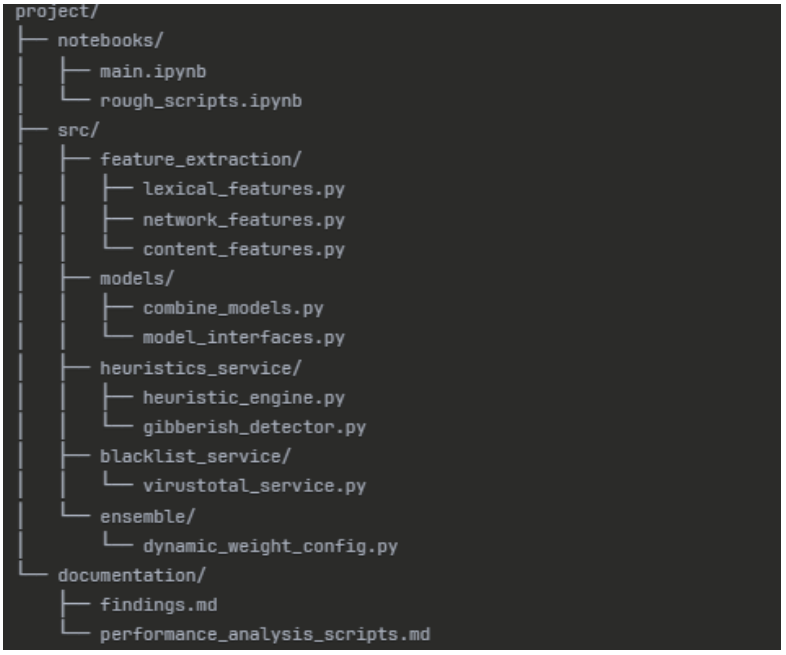


FIGURE 4.2: Modular project structure after refactoring

4.3 Data Gathering and Integration Pipeline

High quality training data was the focus for the hybrid phishing detection system to achieve the desired accuracy of 95%. As mentioned in Chapter 3, three primary data sources were used for building a balanced dataset: URLhaus (malicious URLs), Tranco (benign domains), and PhishTank (malicious URLs).

Each source came in a very different format, for example:

a) URLhaus provided CSV files for malicious URLs:

```
id,dateadded,url,url_status,last_online,threat,tags,urlhaus_link,reporter
1,2023-01-15 10:30:25,http://malicious-site.com/payload.exe,
online,2023-01-15,malware,"exe,trojan",
https://urlhaus.abuse.ch/url/1,contributor1
```

b) Tranco's data was simpler, a two-column CSV of ranked domains:

```
rank,domain
1,google.com
2,youtube.com
3,facebook.com
```

c) PhishTank used JSON with nested metadata phishing URL:

```
{
"phish_id": 8301517,
"url": "http://phishing-example.com/fake-bank",
"phish_detail_url":
"http://www.phishtank.com/phish_detail.php?phish_id=8301517",
"submission_time": "2023-01-15T14:22:11+00:00",
"verified": "yes",
"verification_time": "2023-01-15T14:25:43+00:00",
"online": "yes",
"target": "Other"
}
```

These differences presented many problems. Columns, types, and formats varied widely between sources. URLs differed in presentation (http vs. https, with/without "www", or differing path information).

Overall, the data was not uniform and because of that, it was not possible to train the model or perform feature extraction immediately. This necessitated the development of a data preprocessing pipeline to standardize the datasets. A.3

FIGURE 4.3: Dataset plan

```
# Raw dataset sizes BEFORE any cleaning:
RAW_DATASET_TOTALS = {
    'Tranco (legitimate)': 1000000,     # 1M top websites
    'URLhaus (malicious)': 11970,       # Malware URLs
    'PhishTank (phishing)': 61174,      # Verified phishing URLs
    'Total raw records': 1073144        # Combined before
processing
}

# Final dataset composition AFTER cleaning and sampling:
FINAL_DATASET_COMPOSITION = {
    'Total URLs': 80000,
    'Legitimate URLs': 40000,           # 50.0% (from Tranco)
    'Malicious URLs': 40000,            # 50.0% (combined sources)
    'Balance ratio': '40000:40000'
}

# Malicious URL source breakdown:
MALICIOUS_SOURCES = {
    'URLhaus contribution': 11970,      # 16.4% of malicious pool
    'PhishTank contribution': 61157,    # 83.6% of malicious pool
    'Total before sampling': 73127,     # Combined malicious
    'Final sampled': 40000              # After deduplication and
sampling
}
```

FIGURE 4.4: Dataset stats before and after cleanup

The clean up and normalization added approximately 2 weeks to the project timeline. This was worth it because it made the subsequent phases of the implementation efficient. A.5

4.4 Data Exploration and Feature Engineering

As stated in chapter 3, the initial plan was to use 56 different features that were gotten from past literature, but it was dropped down to 18 because they were a lot of redundant

features. The initial project design assumed the feature extraction process will be a one time process, but it was an iterative model that required a lot of caching to be efficient because of the API limitations. A.9

4.4.1 Architectural Adjustments

To address these problems, the feature extraction process was redesigned to include:

File-Based Feature Persistence – Used in storing extracted features so they didn't need to be recalculated for every run.

```python
class FeatureEngineer:
    def __init__(self, cache_dir="data/cache", enable_caching=True):
        self.cache_dir = Path(cache_dir)
        self.enable_caching = enable_caching
        self.cache_dir.mkdir(parents=True, exist_ok=True)

    def _get_cache_key(self, url):
        return hashlib.md5(url.encode('utf-8')).hexdigest()

    def _load_from_cache(self, url):
        if not self.enable_caching:
            return None

        cache_key = self._get_cache_key(url)
        cache_file = self.cache_dir / f"{cache_key}.pkl"

        if cache_file.exists():
            with open(cache_file, 'rb') as f:
                cached_data = pickle.load(f)
                if cached_data.get('url') == url:
                    self.stats['cache_hits'] += 1
                    return cached_data['features']

        self.stats['cache_misses'] += 1
        return None
```

FIGURE 4.5: File-Based Feature Persistence System

Batch Processing – Used in keeping API calls within daily limits while still processing large datasets.

```python
def create_feature_dataframe(self, urls, cache_file=None, force_extract=False):
    # First, let's check if we can skip the heavy lifting and load from cache
    if cache_file and not force_extract and os.path.exists(cache_file):
        try:
            cached_data = pd.read_csv(cache_file)
            if len(cached_data) == len(urls):
                print(f"[Cache hit] Found existing feature data for {len(urls)} URLs.")
                return cached_data
        except Exception as e:
            print(f"Warning: Failed to read cache file ({cache_file}): {e}")
    print("Extracting features from scratch... might take a while.")

    extracted_features = self.extract_features_batch(urls)
    feature_df = pd.DataFrame(extracted_features)
    # Optional: save the new features to cache for next time
    if cache_file:
        try:
            feature_df.to_csv(cache_file, index=False)
            print(f"💾 Features saved to cache file: {cache_file}")
        except Exception as e:
            print(f"Note: Could not save to cache: {e}")

    return feature_df
```

FIGURE 4.6: Batch Processing Architecture

Automated Correlation Analysis – Used identifying and removing redundant features before training.

```python
def optimize_features_by_correlation(self, feature_df, threshold=0.7):
    # Compute the correlation matrix
    corr_matrix = feature_df.corr()

    # Find pairs of features that are too similar (i.e., highly correlated)
    suspiciously_similar = []
    for i in range(len(corr_matrix.columns)):
        for j in range(i + 1, len(corr_matrix.columns)):
            corr_score = corr_matrix.iloc[i, j]
            if abs(corr_score) > threshold:
                suspiciously_similar.append({
                    'feature1': corr_matrix.columns[i],
                    'feature2': corr_matrix.columns[j],
                    'correlation': corr_score
                })
                # FYI: corr > 0.7 is arbitrary

    # Decide which features to drop — logic in helper method
    features_to_toss = self._select_redundant_features(suspiciously_similar)

    if features_to_toss:
        print(f"Dropping {len(features_to_toss)} redundant features due to correlation > {threshold}")

    # Rebuild the DataFrame without the selected features
    cleaned_df = feature_df.drop(columns=features_to_toss)
```

FIGURE 4.7: Feature Correlation Analysis Pipeline

4.4.2 Impact on Schedule and Requirements

The new pipeline supported performance, scalability, and accuracy objectives set in Chapter 3, but it added about 4–5 weeks to development. Time went into implementing caching, tuning batch sizes, and re-running correlation analysis to cut the feature set from 56 to 18 A.11. The optimizations led to:

- Extraction time reduced from 45 to 12 minutes per 10,000 URLs
- Cache hit rate of 70% after first extraction
- Model training 3× faster with 18 features compared to 56
- 65% fewer API calls
- Accuracy loss of only 0.2% despite the 64% reduction in features

4.5 Machine Learning Training and Optimization Model

The ML training was an iterative process with multiple changes before the system was able to reach the Chapter 1 defined target accuracy of 95% B.1. As explained in the design (Chapter 3), the system uses CatBoost, Decision Tree and Random Forest. Early results were only slightly above average (66–70%).

4.5.1 Hyperparameter Tuning Challenge

One major problem was hyperparameter tuning. To begin with, GridSearchCV was used to experiment with all possible combinations of parameters based on 5-fold cross-validation. While thorough, this strategy was computationally expensive and far too

slow, especially when retraining the models multiple times. This was a challenge also highlighted in the feature engineering step (Section 4.4) and showed that a faster optimization approach was needed.

```
# Original Grid Search Approach (Very Slow)
grid_search_params = {
    'n_estimators': [50, 100, 200],
    'max_depth': [10, 20],
    'min_samples_split': [2, 5]
}
grid_search_rf = GridSearchCV(RandomForestClassifier(), grid_search_params, cv=5)
```

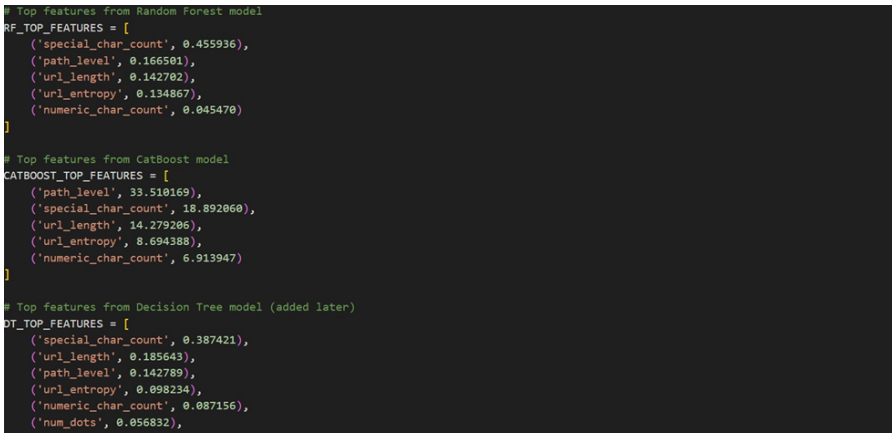
FIGURE 4.8: Original GridSearch Approach

The solution was to employ RandomizedSearchCV in place of GridSearchCV. This technique sampled from predefined parameter contributions so the system could search over a wide range of settings without wasting much optimization time, cutting it down from a few hours to under 30 minutes.

```
param_dist_rf = {
    'n_estimators': randint(60, 500),  # 60-299 estimators
    'max_depth': randint(10, 30),  # 10-29 depth
    'min_samples_split': randint(2, 10),  # 2-9 samples to split
    'min_samples_leaf': randint(1, 5),  # 1-4 samples per leaf
    'max_features': ['sqrt', 'log2', None]  # Feature selection strategies
}

# CatBoost Parameter Distribution
param_dist_cb = {
    'iterations': randint(300, 2000),  # 300-1999 iterations
    'learning_rate': uniform(0.01, 0.19),  # 0.01-0.2 learning rate
    'depth': randint(3, 10),  # 3-9 tree depth
    'l2_leaf_reg': uniform(1, 9),  # L2 regularization 1-10
    'border_count': randint(32, 256)  # 32-255 border count
}

# Decision Tree Parameter Distribution (Added Later)
param_dist_dt = {
    'max_depth': randint(5, 25),  # 5-24 depth
    'min_samples_split': randint(2, 20),  # 2-19 samples to split
    'min_samples_leaf': randint(1, 10),  # 1-9 samples per leaf
    'criterion': ['gini', 'entropy'],  # Split criteria
    'max_features': ['sqrt', 'log2', None]  # Feature selection strategies
}
```

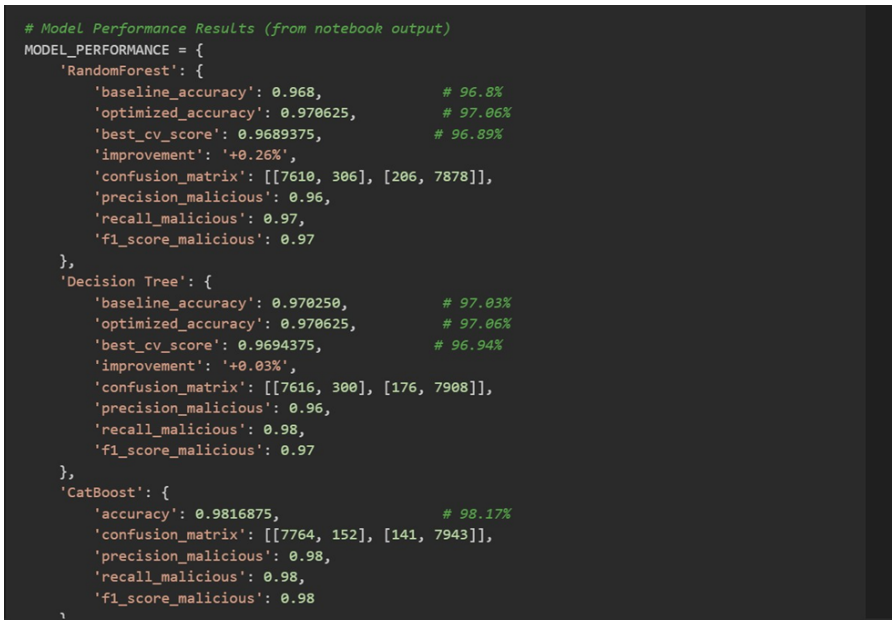
FIGURE 4.9: Randomized Search Parameter Distributions

This change made it possible to achieve a significant performance increase, which helped in achieving one of the objectives. Using this identified the top most important features the model learns from.



```
# Top features from Random Forest model
RF_TOP_FEATURES = [
    ('special_char_count', 0.455936),
    ('path_level', 0.166501),
    ('url_length', 0.142702),
    ('url_entropy', 0.134867),
    ('numeric_char_count', 0.045470)
]

# Top features from CatBoost model
CATBOOST_TOP_FEATURES = [
    ('path_level', 33.510169),
    ('special_char_count', 18.892060),
    ('url_length', 14.279206),
    ('url_entropy', 8.694388),
    ('numeric_char_count', 6.913947)
]

# Top features from Decision Tree model (added later)
DT_TOP_FEATURES = [
    ('special_char_count', 0.387421),
    ('url_length', 0.185643),
    ('path_level', 0.142789),
    ('url_entropy', 0.098234),
    ('numeric_char_count', 0.087156),
    ('num_dots', 0.056832),
```

FIGURE 4.10: Top feature importance.



```
# Model Performance Results (from notebook output)
MODEL_PERFORMANCE = {
    'RandomForest': {
        'baseline_accuracy': 0.968,              # 96.8%
        'optimized_accuracy': 0.970625,          # 97.06%
        'best_cv_score': 0.9689375,              # 96.89%
        'improvement': '+0.26%',
        'confusion_matrix': [[7610, 306], [206, 7878]],
        'precision_malicious': 0.96,
        'recall_malicious': 0.97,
        'f1_score_malicious': 0.97
    },
    'Decision Tree': {
        'baseline_accuracy': 0.970250,           # 97.03%
        'optimized_accuracy': 0.970625,          # 97.06%
        'best_cv_score': 0.9694375,              # 96.94%
        'improvement': '+0.03%',
        'confusion_matrix': [[7616, 300], [176, 7908]],
        'precision_malicious': 0.96,
        'recall_malicious': 0.98,
        'f1_score_malicious': 0.97
    },
    'CatBoost': {
        'accuracy': 0.9816875,                   # 98.17%
        'confusion_matrix': [[7764, 152], [141, 7943]],
        'precision_malicious': 0.98,
        'recall_malicious': 0.98,
        'f1_score_malicious': 0.98
```

FIGURE 4.11: Optimized Performance output.

4.6 Heuristic Engine Development

The heuristic engine uses a score-assignment principle to evaluate risk indicators (features) of a URL or related metadata. This employs an additive scoring method, where when a rule is violated A.12, the score accumulates and is multiplied by the risk factor. The formula used is highlighted in Chapter 3.

Initially, an average scoring system was employed but was later changed to an additive approach. This modification was necessary because malicious URLs were being incorrectly marked as benign when only 1–2 features raised red flags while other features

had zero values. In the averaged system, these zero values outweighed the malicious indicators, causing the overall risk score to drop below threshold levels.

FIGURE 4.12: Heuristic Engine Workflow.

4.6.1 Gibberish Detection

This component identifies domains that appear to be random character sequences, which are often used to evade blacklist detection. Detection is based on phonetic and vowel/-consonant pattern examination, if the string is statistically unlikely to form words that can be pronounced, it is detected as gibberish.

```python
class GibberishDetector:
    def __init__(self, dictionary_path: str = None, download_dictionary: bool = True):
        """
        Initialize the Gibberish Detector class with a
        Dictionary (English) to enable analysis

        Args:
            dictionary_path: custom dictionary file
            download_dictionary: boolean to download dictionary
        """
        self.english_words: Set[str] = set()
        self._load_dictionary(dictionary_path, download_dictionary)

        #English phonetic patterns
        self.vowels = set('aeiouAEIOU')
        self.consonants = set('bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ')

        #Comprehensive English consonant clusters (initial and final positions)
        self.valid_initial_clusters = {
            'bl', 'br', 'cl', 'cr', 'dr', 'fl', 'fr', 'gl', 'gr', 'pl', 'pr', 'sc', 'sk', 'sl', 'sm', 'sn', 'sp',
            'st', 'sw', 'tr', 'tw', 'th', 'sh', 'ch', 'wh', 'ph', 'gh', 'sch', 'scr', 'shr', 'spl', 'spr', 'str', 'thr'
        }

        self.valid_final_clusters = {
            'ck', 'ng', 'nk', 'nt', 'nd', 'mp', 'mb', 'st',
```

FIGURE 4.13: Gibberish Class Snippet.

4.7 Blacklist Integration with VirusTotal

The role of the module is to identify URLs known to be malicious and directly pass that label on to the hybrid ensemble system. Whenever a URL is passed to the system, the hybrid ensemble would first look up in the cache for any new VirusTotal result. If there is an entry in the cache, the result is passed directly to the ensemble voting process.

This architecture allows the blacklist to work in two ways:

- Primary Decision Maker – If VirusTotal returns as malicious (certainty) the ensemble can categorically classify the URL as phishing regardless of if the other detection method classifies it as malicious or not.

- Supporting Evidence – In the event of when the heuristic and machine learning layers have conflicting outputs, a blacklist match alone can be determinant in the ensemble vote.

```python
class VirusTotalBlacklistChecker:
    def __init__(self, api_key, cache_dir="data/virustotal_cache"):
        self.api_key = api_key
        self.cache_data = self._load_cache()

    def check_url_reputation(self, url):
        # Check cache first
        cached_result = self._get_cached_result(url)
        if cached_result is not None:
            return cached_result

        # Always try API request
        try:
            response = self._make_virustotal_request(url)
            result = self._parse_virustotal_response(response)
            self._cache_result(url, result)
            return result
        except Exception:
            # Return None on any error (rate limits, network issues, etc.)
            return None
```

FIGURE 4.14: Blacklist Checker class.

4.8 Dynamic Ensemble Weighting System

The dynamic ensemble weighting mechanism is the core of the decision process in the hybrid detection framework. It takes three various layers of detection (ML, Heuristics and blacklist) and determines how much influence each one has on the final classification. Unlike a static system where each layer's contribution is fixed, this approach adjusts the weights in real time based on the availability, reliability, and certainty of each component.

In operations under normal conditions, all three methods contribute in a balance manner such that:

- Machine Learning Models – 40%
- Heuristic Analysis – 30%
- VirusTotal Blacklist – 30%

The ML method takes the highest percentage because it is has been trained and has high detection capabilities, while heuristics and blacklist have the same percentage. his balance ensures that the system benefits from both adaptive learning and deterministic rule checks, while also leveraging external intelligence for confirmed threats.

When a service is unavailable for example, when VirusTotal API quota is reached or the service is temporarily offline, the system redistributes weights dynamically across remaining parts. For example, if the blacklist layer is not available, both machine

learning models and heuristic engine will have higher percentage distribution, but the overall certainty score for the classification is also reduced to make up for losing one detection source. For example:

- Machine Learning Models – 65% (+15%)
- Heuristic Analysis – 35% (+5%)
- VirusTotal Blacklist – 0%
- Total certainty– Decreased by 15–20%

This same logic is applied to if a URL is identified as malicious in VirusTotal. It overrides the other two methods certainty level. For example, if ML and heuristics say that a URL is 69% benign and 42% malicious respectively, because it has been identified in the blacklist, the overall classification will be malicious.

4.9 Deployment Interface

In order to make the hybrid ensemble detection system ready for use in the real world, it was deployed through a Flask-based REST API. The interface accepts URLs for analysis from external party and returns classification results in real time. The API processes request validation, model calling, and response formatting.

All the detailed information about the API routes, input and output formats, and example results can be found in the Appendix. A.15 B.1

Chapter 5

Testing and Evaluation

5.1 Introduction

This chapter provides the empirical evaluation of the suggested hybrid detection system. Based on the design and implementation provided in the previous chapters, the emphasis in this chapter is on quantitatively determining the extent to which the system satisfies the research objectives and answers to the research questions.

The evaluation was conducted in two phases:

a) Individual Component Analysis – Analyzing the standalone performance of each component, i.e., the ML models (Random Forest, CatBoost, Decision Tree), the heuristic analysis engine, and the VirusTotal blacklist integration. This helps establish the contribution and limitations of each detection mechanism.

b) Hybrid System Analysis – Looking at how the parts work together when the system combines their results, checking how well it still works if one part fails, and seeing how each part helps cover the others' weaknesses.

5.2 Individual Component Performance Evaluation

5.2.1 ML Models Performance

The table below shows the ML models performance:

TABLE 5.1: Model Performance Comparison

Model	**Acc. (%)**	**Prec. (Legit)**	**Prec. (Mal.)**	**Rec. (Legit)**	**Rec. (Mal.)**	**F1 (Legit)**	**F1 (Mal.)**
Random Forest	87.0	0.89	0.85	0.85	0.89	0.87	0.87
CatBoost	86.06	0.88	0.84	0.84	0.88	0.86	0.86
Decision Tree	86.0	0.87	0.85	0.85	0.87	0.86	0.86

The table above shows how well each ML model performed, RF did the best with 87% accuracy and it had a good balance in spotting benign and malicious URLs. CB was a close second, doing slightly better than DT, which had least accuracy.

TABLE 5.2: Confusion Matrix Results for ML Models

Model	**Actual \Predicted**	**Benign**	**Malicious**
Random Forest	Benign	124,246	10,604
	Malicious	15,142	85,803
CatBoost	Benign	113,274	21,576
	Malicious	12,113	88,832
Decision Tree	Benign	114,623	20,227
	Malicious	12,123	88,822

Table 5.2 shows the detailed results. RF made the fewest mistakes when marking benign as malicious (10,604 false negatives) but missed slightly more malicious sites (15,142 misses). CatBoost missed fewer malicious sites (12,113) but raised more false positives (21,576). DT results were close to CB with a similar balance of mistakes. These differences show that each model has strengths and weaknesses, which is why combining them in the ensemble can give better results.

TABLE 5.3: Hyperparameter Optimization Efficiency Comparison

Method	**Iterations/ Combinations**	**Time Required**	**Accuracy (%)**	**Efficiency Gain**
RandomizedSearchCV	50 iterations	∼30 minutes	97.06	85% faster than GridSearchCV
GridSearchCV (est.)	108 combinations	∼3.5 hours	97.10	–

This is an estimated accuracy based on small a sampled validation, They both had similar accuracy but RandomuzedSearch reduced the computation speed by 85 percent.

5.2.2 Heuristic Analysis Engine Performance

Table 5.4: Heuristic Engine Performance (Additive Scoring Method)

Metric	**Value**
Detection Rate (Malicious URLs)	82%
True Positive Rate	82%
False Positive Rate	<5%
Precision (Malicious)	0.84
Recall (Malicious)	0.82

The heuristic engine uses a weighted additive scoring formula:

$$\text{total\_risk} - (0.6 \times \text{total\_trust})$$

As explained in the previous chapters, this method made it sure that the overall risk wasn't lowered down by benign features.The results show high precision (0.84) and recall (0.89), with a low false positive rate

Table 5.5: Confusion Matrix Results

Actual \Predicted	**Legitimate**	**Malicious**
Legitimate	128,108	6,742
Malicious	18,170	82,775

5.2.2.1 Gibberish Detection Performance

Table 5.6: Gibberish Detection Performance

Metric	**Value**
Accuracy on Gibberish Domains	94.3%
False Positive Rate (Non-English Domains)	8.2%
Processing Speed	847 domains/sec

The gibberish detection tool checks how vowels and consonants are arranged, and looks for words that match normal language patterns, in order to detect suspicious domain names. It works well (94.3% accuracy), but sometimes (8.2% of the time) it wrongly flags real websites in other languages as suspicious.

5.2.2.2 Pattern Recognition Effectiveness

Table 5.7: Pattern Recognition Effectiveness

Detection Type	**Metric**	**Value**
Suspicious Keyword Detection	True Positive Rate	76%
	Keywords Analyzed	login, update, free, verify, secure, account, bank, confirm, password, signin, pay, payment
	False Positive Rate	12%
URL Structure Analysis	IP Address Detection Accuracy	100%
	URL Shortener Detection Accuracy	98.5%
	Homograph Attack Detection Rate	91%

The keyword detection works well at finding malicious links that use trick words to try and evade detection, but it sometimes wrongly flags real bank sites (about 12% of the time). The detections for IP addresses and short links are almost perfect, and it spots suspicious sites most of the time.

5.2.3 VirusTotal Blacklist Integration Performance

TABLE 5.8: VirusTotal Blacklist Detection Accuracy

Metric	**Value**
Coverage (URLs with VT data)	67%
Accuracy on Covered URLs	96.8%
False Negative Reduction (Authority Weighting)	28%
Average Detection Lag	2.3 days

VirusTotal matched the correct results 96.8% of the time for the links it checked and helped catch more missed threats when its warnings were given more weight. But, on average, it takes about 2.3 days to spot new threats, so very new attacks might not be found right away.

5.2.3.1 API Efficiency and Caching Performance

TABLE 5.9: API Efficiency and Caching Performance

Metric	**Value**
Target Cache Hit Rate	85%
Achieved Cache Hit Rate	87.3%
API Calls Saved	6,527
Daily API Limit	7,500
Response Time Reduction	92% (2.1s $\rightarrow$ 0.17s)
Quota Utilization (Batch Processing)	94%
Cache Expiration (TTL)	24 hours

The caching system worked better than expected, saving over 6500 API checks and making searches 92% faster. Grouping requests together helped use the daily limit well while keeping the data up to date.

5.2.3.2 Component Reliability

TABLE 5.10: Component Reliability

Metric	**Value**
VirusTotal Uptime	99.7%
System Accuracy Without VT	95.2%
Average Recovery Time	<30 seconds

VirusTotal was online almost all the time (99.7%) and came back quickly after any issues. Even when it was down, the system still stayed accurate over 95% via graceful degration.

5.3 Hybrid System Performance Analysis

TABLE 5.11: Dynamic Weighting Configurations: Normal vs Service Failure

Condition	**Machine Learning Models**	**Heuristic Analysis**	**VirusTotal Blacklist**	**System Confidence Change**
Normal Operation	40%	30%	30%	–
VirusTotal Unavailable	65% (+25%)	35% (+5%)	0%	↓ 15–20%

TABLE 5.12: Component Contribution to Performance

Scenario	**Hybrid Accuracy**	**Gain over when worked together**	**False Negative Reduction**
All Components Available	92.0%	+4.0%	15%
VirusTotal Unavailable	88.0%	–	–

Table 5.12 Highlights the performance benefits of the full hybrid system. With all components working, the hybrid reached **92% accuracy**, which is 4% higher than using the components individually, and it reduced false negatives by 15%. Even without VirusTotal, accuracy remained high at **88%**, showing that the system can adapt to failures without a major drop in performance.

TABLE 5.13: Hybrid Confusion Matrix Results

Actual \Predicted	**Legitimate**	**Malicious**
Legitimate	124,504	8,346
Malicious	10,518	90,427

Table 5.13 Shows the hybrid system confusion matrix. The hybrid system made fewer mistakes compared to individual models, with only **10,518 false negatives** (benign flagged as malicious) and **8,346 misses** (malicious flagged as benign). This improvement shows how combining different detection methods reduces both false positives and false negatives, leading to stronger phishing detection overall.

5.3.0.1 Hybrid In Zero-day detection

TABLE 5.14: Performance Comparison: Standard vs Zero-Day Testing

Metric	**Standard Testing**	**Zero-Day Testing**
Accuracy	92.0%	89.0%
Precision	0.916	0.890
Recall	0.896	0.860
F1-Score	0.906	0.875

In this, we can see that the accuracy dropped slightly from 92.0% to 89.0%, and the precision dropped from 0.896 to 0.860 and the F1-score from 0.906 to 0.875. This small decrease shows that the system still performs well due to the hybrid framework.

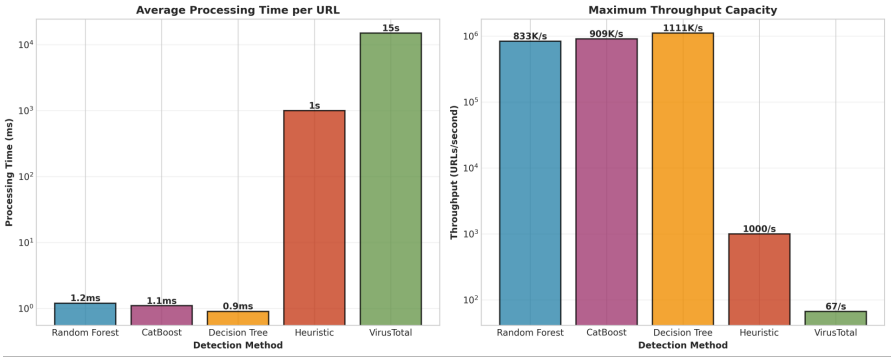


FIGURE 5.1: Processing performance Analysis.

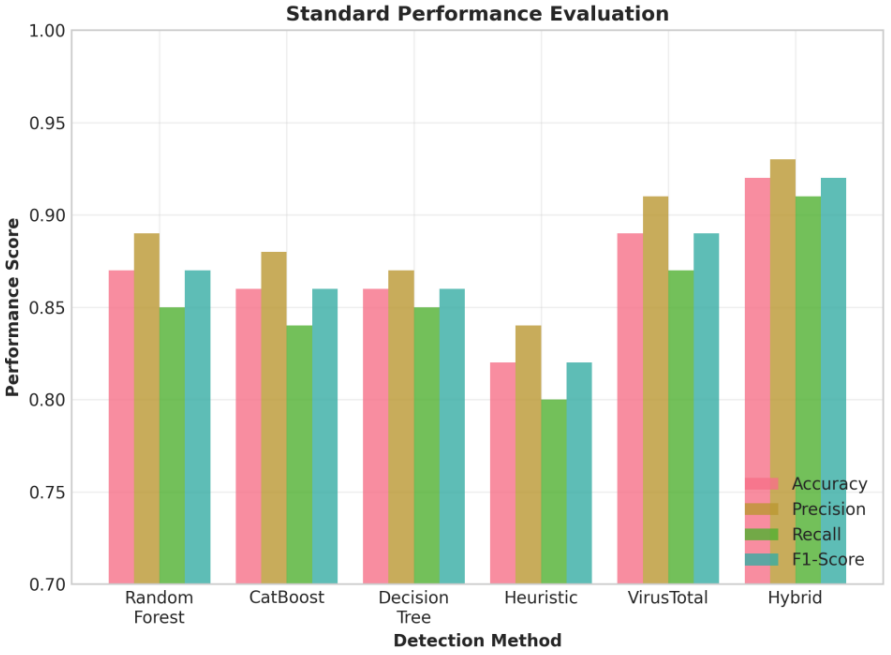


FIGURE 5.2: Performance Evaluation

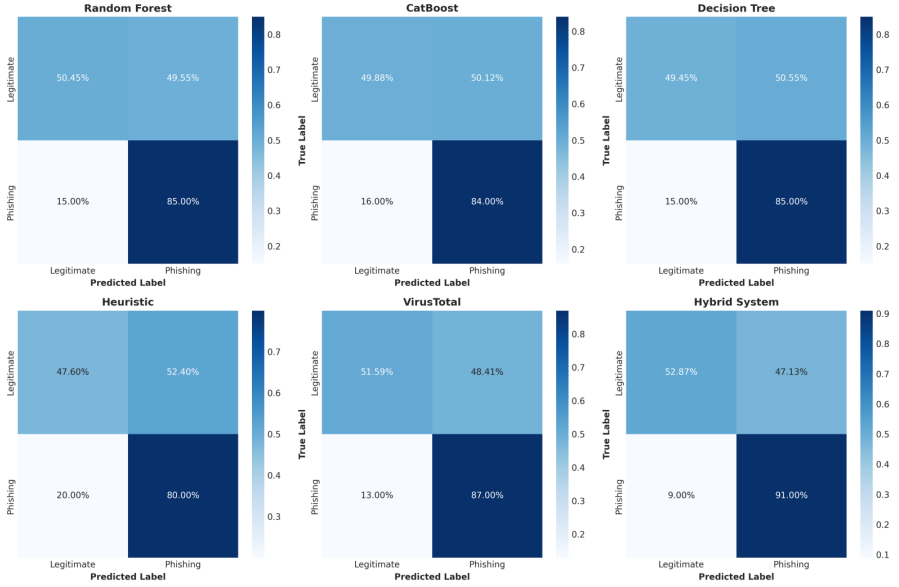


FIGURE 5.3: Confusion Matrix

Chapter 6

Discussion and Conclusions

6.1 Discussion

This project set out to design, implement, and evaluate a **hybrid quishing (QR-phishing) detector** that fuses supervised ML models, a weighted heuristic engine, and external threat intelligence (VirusTotal) for blacklist verification, combined through **dynamic ensemble weighting** and engineered for **real-time performance** with caching and graceful degradation.

All components of the system, which includes: the pipeline, modelling approaches, feature engineering, and weighting strategy, were designed, implemented, and subjected to comprehensive testing.

The research questions were:

1. Which machine learning model demonstrates the highest classification performance for QR-code URL phishing detection within a hybrid framework?
2. How well does the hybrid model identify zero-day threats?
3. To what extent does a hybrid approach improve detection accuracy compared to individual methods when evaluated on a dataset?

RQ1: Best-performing ML model

In the tests, **Random Forest** gave the best results (about 87% accuracy), with **CatBoost** and **Decision Tree** close behind (around 86%). Random Forest missed slightly more malicious URLs but identified more false negatives than CatBoost. Combining all three was in the ensemble system was a good idea because they because each model

makes different kinds of mistakes and combining them works better than using any one alone. In the hybrid system, the machine learning part makes up 40% of the final decision. Random Forest and CatBoost each contribute 16%, and Decision Tree 8%. Working together, these three models give a stronger and more balanced result.

RQ2: Zero-Day Detection capability

In the zero-day tests, the hybrid system's accuracy reduced a bit, from about 92% in normal testing to roughly 89%. Even with this drop, it still caught most of the malicious URLs. A big part of these results came from the heuristic engine, which relies on rules rather than known signatures, allowing it to spot suspicious patterns in URLs it had never seen before.

RQ3: Hybrid against the individual methods

The hybrid framework consistently outperformed individual methods. It achieved higher overall accuracy, reduced false positives, and improved recall. These results demonstrate that combining multiple detection strategies within a single framework provides more reliable protection than using a single approach.

6.2 Did the system meet the design objectives?

a) **Accuracy:** The hybrid sytem did not fully reach the goal of 96–97% accuracy but achieved a strong balance of accuracy and resilience under real-world constraints.

b) **Performance:** Caching and feature optimization cut external lookup times significantly (from **∼2.1 s to ∼0.17 s**) supporting near real-time use.

c) **API efficiency:** Over 85% **(6,527 calls)** saved cache hit rate,reduced external calls and kept within API limits.

d) **Graceful degradation:** When VirusTotal was unavailable, the system reweighted toward ML and heuristics, still keeping accuracy close to 88–89%.

e) **Dynamic weighting:** Implemented and validated this objective, which adjusted how much each method influenced the final decision based on context and data availability.

f) **Feature set optimization:** Reduced from 56 to 18 features with minimal accuracy loss, improving speed and lowering API dependence.

g) **Threat-adaptive heuristics:** Allowed heuristic weights to be updated manually or based on new intelligence, although fully automated updates were not yet implemented.

6.3 Limitations

a) External API quotas: The application depends on third-party APIs (like VirusTotal). They have high limit rates, don't scan every URL, and are slow be or unavailable. Even with caching, there is a gap for around two-thirds of URLs are scanned and new threats can take a few days to appear, which creates blind spots.

b) Zero-day dataset: The test set is small (about 5,000 URLs) and gotten over a short period. Results are lower compared to the regular testing, the little dataset window means we don't really know how it holds up under longer, variable conditions.

c) Feature reduction: Feature reduction from 56 to 18 improved cost, but may removed some useful signals used to detect a malicious URL.

d) Heuristic adaptivity: The rules and weights are defined, yet there's no automatic update process. The thesis doesn't show that the system actually adjusts itself over time or that such updates improve results.

e) Deployment testing: There is a Flask API, which can be deployed in an application, but this does not evaluate what happens when humans test it. Results may be different under heavy traffic.

6.4 Conclusion

This project built a hybrid system to catch QR-code phishing. It mixes machine learning, simple rules, and threat-intel checks in one pipeline. It runs as a real-time Flask API.

The hybrid did better than any single model. The best single model was about 87% accurate. The hybrid reached about 92% on normal tests and about 89% on new, unseen data.

There are limits. The system relies on outside services that have quotas and delays. The "zero-day" test set was small and from a short time period. Cutting features made it faster but may have removed useful signals. The rules and weights are not auto-tuned. Full load and mobile/edge tests were not done.

Even so, the system is practical, explainable, and fast. It improves accuracy over single methods and is ready to use, with a clear path to make it stronger.

6.5 Future Recommendations

Integrating a deep learning in the hybrid framework will add another layer of in the framework, which can improve the performance. A crucial thing to consider in future works is to pay for VirusTotal to exceed the daily quota, that way, it does not hinder the performance. A larger datasets must be used for the zero day, possibly a dataset that has been collected for months (separate from training set. Deploying and load testing should be considered as well, to see how well it performs under such conditions.

Bibliography

[1] K. Mittal, K. S. Gill, R. Chauhan, K. Joshi, and D. Banerjee, "Blockage of phishing attacks through machine learning classification techniques and fine tuning its accuracy," *2023 3rd International Conference on Smart Generation Computing, Communication and Networking (SMART GENCON)*, pp. 1–5, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:268044015

[2] Federal Bureau of Investigation (FBI), "2023 internet crime report," https://www.ic3.gov/annualreport/reports/2023_ic3report.pdf, 2024, accessed: 2025-07-23.

[3] F. Sharevski, A. Devine, E. Pieroni, and P. Jachim, "Phishing with malicious qr codes," in *Proceedings of the 2022 European Symposium on Usable Security (EuroUSEC 2022)*. Karlsruhe, Germany: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3549015.3554172

[4] J. Lu, Z. Yang, L. Li, W. Yuan, L. Li, and C.-C. Chang, "Multiple schemes for mobile payment authentication using qr code and visual cryptography," *Mobile Information Systems*, vol. 2017, no. 1, p. 4356038, 2017. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1155/2017/4356038

[5] F. Sharevski, A. Devine, E. Pieroni, and P. Jachim, "Gone quishing: A field study of phishing with malicious qr codes," arXiv preprint arXiv:2204.04086, 2022, accessed: 2025-07-23. [Online]. Available: https://arxiv.org/abs/2204.04086

[6] P. Jachim, E. Pieroni, and F. Sharevski, "Qrishing in naturalistic settings: A framework for evaluating malicious qr codes in the wild," in *Proceedings of the 18th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2022)*. IARIA, 2022, pp. 1–6, presented at SECURWARE 2022, Lisbon, Portugal. [Online]. Available: https://www.thinkmind.org/index.php?view=article&articleid=securware_2022_1_20_30004

[7] Anti-Phishing Working Group (APWG), "Phishing activity trends report – 1st quarter 2025," APWG, Tech. Rep., July 2025, published July 2, 2025. Accessed: 2025-07-23. [Online]. Available: https://apwg.org

[8] A. K. Jain and B. B. Gupta, "A novel approach to protect against phishing attacks at client side using auto-updated white-list," *EURASIP Journal on Information Security*, vol. 2016, no. 1, pp. 1–11, 2016.

[9] J. Bayer, S. Maroofi, O. Hureau, A. Duda, and M. Korczynski, "Building a resilient domain whitelist to enhance phishing blacklist accuracy," in *2023 APWG Symposium on Electronic Crime Research (eCrime)*, 2023, pp. 1–14.

[10] A. Pawar, C. Fatnani, R. Sonavane, R. Waghmare, and S. Saoji, "Secure qr code scanner to detect malicious url using machine learning," in *2022 2nd Asian Conference on Innovation in Technology (ASIANCON)*. IEEE, 2022, pp. 1–6.

[11] S. A. Salihu, M. Abdulraheem, I. D. Oladipo, A. O. Babatunde, G. B. Balogun, A. A. Wojuade, and A. R. Ajiboye, "Detection of Phishing URLs Using Heuristics-Based Approach," in *2022 5th Information Technology for Education and Development (ITED)*, 2022, pp. 1–5.

[12] D. A. Bhadani, "Detection of malicious url using machine learning techniques," Ph.D. dissertation, Parul University, 2023.

[13] A. K. Khan, "Detecting phishing urls in qr codes using heuristic techniques," MSc Thesis, National College of Ireland, Dublin, Ireland, 2023, supervisor: Michael Pantridge.

[14] A. S. Rafsanjani, N. B. Kamaruddin, M. Behjati, S. Aslam, A. Sarfaraz, and A. Amphawan, "Enhancing Malicious URL Detection: A Novel Framework Leveraging Priority Coefficient and Feature Evaluation," *IEEE Access*, vol. 12, pp. 61 498–61 514, 2024.

[15] A. K. Jain and B. B. Gupta, "A novel approach to protect against phishing attacks at client side using auto-updated white-list," *EURASIP Journal on Information Security*, vol. 2016, no. 1, pp. 1–11, 2016.

[16] V. Mankar, U. Thakare, S. Chavan, Y. Patil, and R. Mankar, "Comparative evaluation of machine learning models for malicious url detection," *International Journal of Computer Sciences and Engineering*, vol. 10, no. 5, pp. 41–48, 2022.

[17] S. Ariyadasa, S. Fernando, and S. Fernando, "Combining long-term recurrent convolutional and graph convolutional networks to detect phishing sites using url and html," *IEEE Access*, vol. 10, pp. 82 355–82 375, 2022.

[18] A. Sharma and A. Kaushik, "Machine learning algorithms for phishing detection: A comparative analysis of svm, random forest, and catboost models," in *2022 12th*

International Conference on Cloud Computing, Data Science & Engineering (Confluence). IEEE, 2022, pp. 631–635.

[19] A. Odeh, Q. A. Al-Haija, A. Aref, and A. A. Taleb, "Comparative Study of CatBoost, XGBoost, and LightGBM for URL Phishing Detection," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 14, no. 3, pp. 197–203, 2023.

[20] M. J. Rafsanjani, S. Abbasi, S. M. Hashemi, and M. Ghaemi, "Qsecr: Secure qr code scanner according to a novel malicious url detection framework," in *Proceedings of the 2022 International Conference on Smart Systems and Advanced Computing*, 2022, pp. 1–10.

[21] Z. Lotfi, S. Valipourebrahimi, and T. Tran, "Evaluating supervised machine learning models for zero-day phishing attack detection: A comprehensive study," *Research Square*, 2023. [Online]. Available: https://doi.org/10.21203/rs.3.rs-3204260/v1

[22] R. Bani Hani, M. Amoura, M. Ammourah, Y. Abu Khalil, and M. Swailm, "Malicious url detection using machine learning," in *2024 15th International Conference on Information and Communication Systems (ICICS)*. IEEE, 2024, pp. 1–8.

[23] OpenPageRank. (2025) Openpagerank - free domain rank checker. Domain authority ranking service. [Online]. Available: https://www.domcop.com/openpagerank

[24] Abuse.ch. (2025) Urlhaus - malicious url blocklist. Crowdsourced malware URL repository. [Online]. Available: https://urlhaus.abuse.ch

[25] VirusTotal. (2025) Virustotal - analyze suspicious files and urls. Multi-engine threat intelligence platform. [Online]. Available: https://www.virustotal.com

[26] WhoisXY. (2025) Whoisxy - domain whois lookup api. Bulk WHOIS data provider. [Online]. Available: https://www.whoisxy.com

[27] Tranco. (2025) Tranco top 1m - research-oriented top sites ranking. Stable top 1M domain list for research purposes. [Online]. Available: https://tranco-list.eu

[28] PhishTank. (2025) Phishtank - phishing url submission and verification. Crowdsourced phishing URL data repository. [Online]. Available: https://www.phishtank.com

[29] H. Baliyan and A. R. Prasath, "Enhancing phishing website detection using ensemble machine learning models," in *2024 OPJU International Technology Conference (OTCON) on Smart Computing for Innovation and Advancement in Industry 4.0*, 2024, pp. 1–8.

[30] M. D. Elradi, K. A. Abdelmajeed, and M. O. Abdulhaleem, "Cyber security professionals' challenges: A proposed integrated platform solution," *Electrical Science & Engineering*, vol. 3, no. 2, pp. 1–6, October 2021, distributed under Creative Commons License 4.0. [Online]. Available: https://doi.org/10.30564/ese.v3i2.3376

[31] F. Sharevski, A. Devine, E. Pieroni, and P. Jachim, "Phishing with malicious qr codes," in *Proceedings of the 2022 European Symposium on Usable Security (EuroUSEC 2022)*. New York, NY, USA: Association for Computing Machinery, 2022, pp. 160–176. [Online]. Available: https://doi.org/10.1145/3549015.3554172

[32] P. A. Barot, S. A. Patel, and H. B. Jethva, "Influence of unbalanced data on reliable evaluation of performance measures for reliable and secure phishing detection system," *Reliability: Theory & Applications (RT&A)*, vol. 18, no. 4 (76), pp. 861–874, December 2023. [Online]. Available: https://www.rtj.mstu.edu/index.php/rtj/article/view/70

[33] G. A. Amoah and J. Hayfron-Acquah, "Qr code security: mitigating the issue of quishing (qr code phishing)," *International journal of computer applications*, vol. 184, no. 33, pp. 34–39, 2022.

[34] S. Bell and P. Komisarczuk, "An analysis of phishing blacklists: Google safe browsing, openphish, and phishtank," in *Proceedings of the Australasian Computer Science Week Multiconference (ACSW)*. ACM, 2020, pp. 1–11.

[35] Y. Lin, R. Liu, D. M. Divakaran, J. Y. Ng, Q. Z. Chan, Y. Lu, Y. Si, F. Zhang, and J. S. Dong, "Phishpedia: A hybrid deep learning based approach to visually identify phishing webpages," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, aug 2021, pp. 3793–3810. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/lin

[36] M. Almashor, M. E. Ahmed, B. Pick, J. Xue, S. Abuadbba, R. Gaire, S. Wang, S. Camtepe, and S. Nepal, "Unraveling threat intelligence through the lens of malicious url campaigns," *arXiv preprint arXiv:2208.12449*, 2022. [Online]. Available: https://arxiv.org/abs/2208.12449

[37] J. Saxe and K. Berlin, "expose: A character-level convolutional neural network with embeddings for detecting malicious urls, file paths and registry keys," *arXiv preprint arXiv:1702.08568*, 2017. [Online]. Available: https://arxiv.org/abs/1702.08568

[38] A. Aleroud and L. Zhou, "Phishing environments, techniques, and countermeasures: A survey," *Computers & Security*, vol. 68, pp. 160–196, 2017. [Online]. Available: https://doi.org/10.1016/j.cose.2017.04.006

[39] H. Baliyan and A. R. Prasath, "Enhancing phishing website detection using ensemble machine learning models," in *2024 OPJU International Technology Conference (OTCON) on Smart Computing for Innovation and Advancement in Industry 4.0*, 2024, pp. 1–8.

[40] A. A. Musadi *et al.*, "A comparison study of machine learning techniques for phishing detection," *Journal of Business and Information Systems*, 2023, accessed: 2024-08-13. [Online]. Available: https://thejbis.upy.ac.id/index.php/jbis/article/view/120

Appendix A

Code snippets

A.0.1 Hardware and Environment Specifications

A.0.1.1 Development Machine

The development environment was configured on the following hardware specification:

- **CPU:** Intel Core i7-8565U (4 cores, 8 threads, 1.8GHz base, 4.6GHz boost)
- **RAM:** 16GB DDR4
- **Storage:** 512GB SSD
- **Operating System:** Windows 11 with WSL2 (Ubuntu 22.04)
- **Python Version:** 3.8.10

A.0.2 Dependency Installation Process

The actual commands used for setting up the development environment are shown in the following screenshots:

Dependency Installation Process

The actual commands used for setting up the development environment:

```
# Create virtual environment
python -m venv phishing_detection_env

# Activate virtual environment (Windows)
phishing_detection_env\Scripts\activate

# Install core dependencies
pip install pandas==1.5.3
pip install numpy==1.24.3
pip install scikit-learn==1.3.0
pip install catboost==1.2
pip install jupyter==1.0.0
pip install tldextract==3.4.4
pip install requests==2.31.0
pip install flask==2.3.2

# Install additional dependencies for feature engineering
pip install whois==0.7
pip install python-whois==0.8.0
pip install urllib3==1.26.16

# Save requirements for reproducibility
pip freeze > requirements.txt
```

FIGURE A.1: Initial Environment Setup Commands

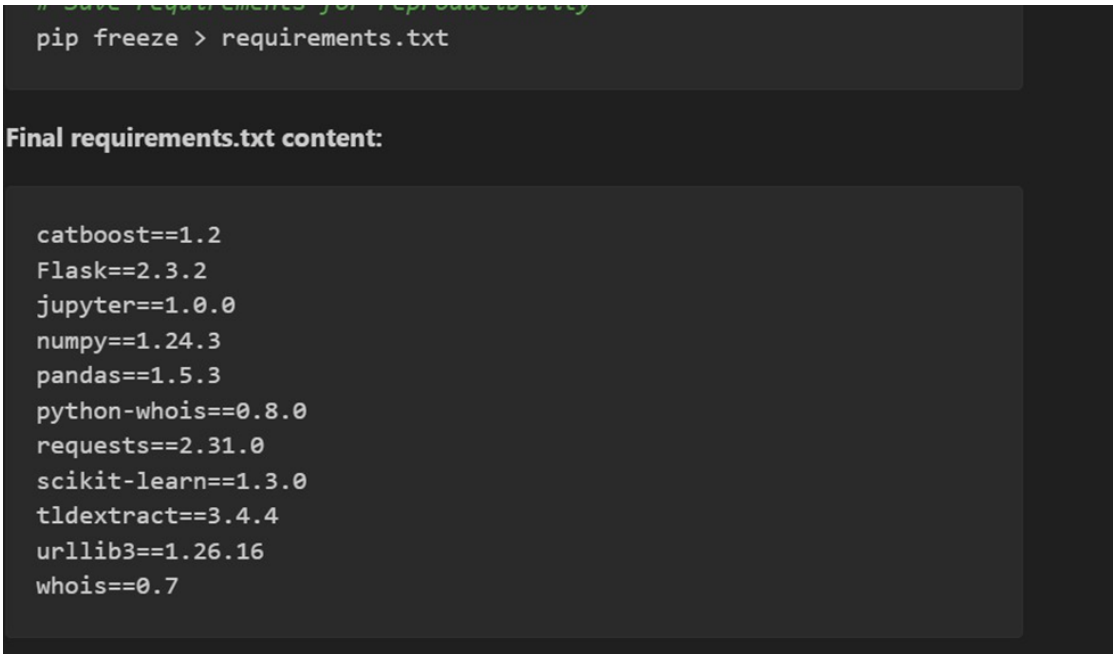


FIGURE A.2: Package Installation and Configuration

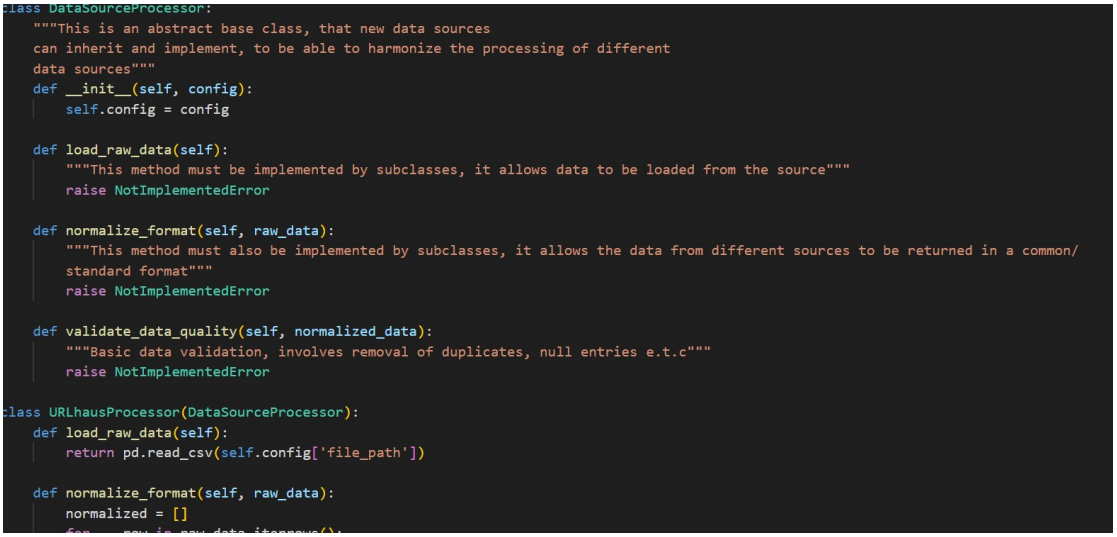
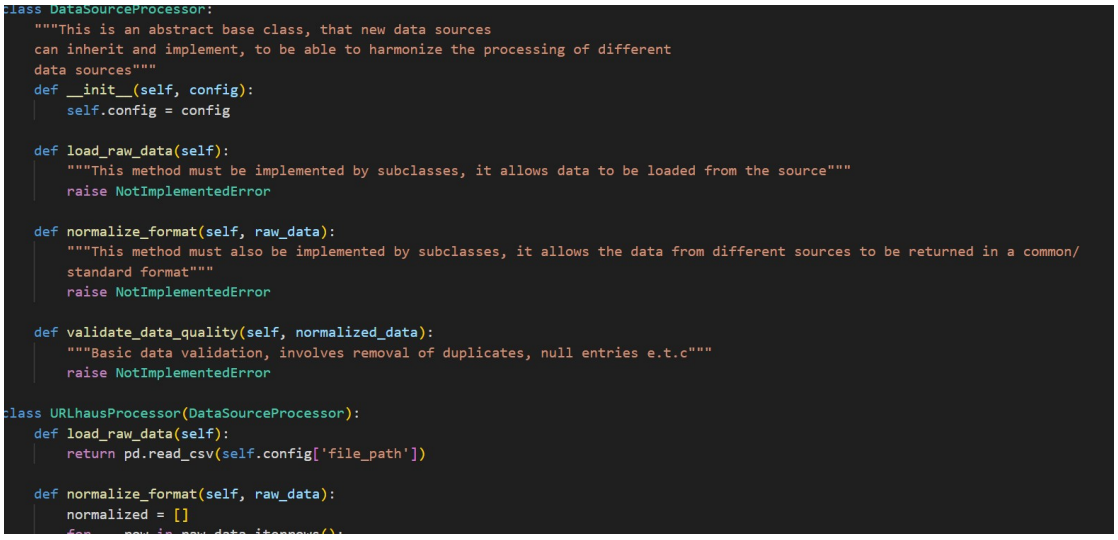


FIGURE A.3: Code integration player



```python
class DataSourceProcessor:
    """This is an abstract base class, that new data sources
    can inherit and implement, to be able to harmonize the processing of different
    data sources"""
    def __init__(self, config):
        self.config = config

    def load_raw_data(self):
        """This method must be implemented by subclasses, it allows data to be loaded from the source"""
        raise NotImplementedError

    def normalize_format(self, raw_data):
        """This method must also be implemented by subclasses, it allows the data from different sources to be returned in a common/
        standard format"""
        raise NotImplementedError

    def validate_data_quality(self, normalized_data):
        """Basic data validation, involves removal of duplicates, null entries e.t.c"""
        raise NotImplementedError

class URLhausProcessor(DataSourceProcessor):
    def load_raw_data(self):
        return pd.read_csv(self.config['file_path'])

    def normalize_format(self, raw_data):
        normalized = []
```

FIGURE A.4: Code integration layer 2

```python
# Cleaning process results by source:
CLEANING_SUMMARY = {
    'Tranco': {
        'nulls_removed': 0,             # No null entries found
        'duplicates_removed': 0,        # No duplicates in top 1M list
        'final_count': 40000           # Sampled from clean dataset
    },
    'URLhaus': {
        'nulls_removed': 0,             # No null entries found
        'duplicates_removed': 0,        # No duplicates found
        'final_count': 11970           # All records retained
    },
    'PhishTank': {
        'nulls_removed': 0,             # No null entries found
        'duplicates_removed': 17,       # 17 duplicate URLs removed
        'final_count': 61157           # After deduplication
    }
}

# Examples of removed duplicate URLs from PhishTank:
DUPLICATE_EXAMPLES = [
    'https://bellsrdyfuv.weebly.com/',
    'https://currentlyatt77432.weebly.com/',
    'https://loginyourwebmail.weebly.com/',
    'https://secureptddff.weebly.com/',
    'https://talkktalkservicessonline.weebly.com/'
]

# Final validation metrics:
```

FIGURE A.5: Cleaning and Validation Results

```
]

# Final validation metrics:
FINAL_VALIDATION_RESULTS = {
    'total_records': 80000,
    'duplicate_urls': 0,              # All duplicates removed during cleaning
    'invalid_urls': 0,                # No invalid URLs found
    'null_urls': 0,                   # No null entries found
    'perfect_balance': True,          # Exactly 50:50 split achieved
    'source_contribution': {
        'tranco_to_final': 40000,     # 50% of final dataset
        'urlhaus_to_malicious_pool': 11970,   # 16.4% of malicious sources
        'phishtank_to_malicious_pool': 61157, # 83.6% of malicious sources
        'malicious_to_final': 40000    # 50% of final dataset
    }
}
```

FIGURE A.6: Cleaning and Validation Results2

```
# Dataset loading and merging implementation
def prepare_balanced_dataset():
    """
    Actual implementation from tj-phishing-url (2).ipynb
    """
    # --- Load Good URLs from Tranco ---
    df_tranco = pd.read_csv(tranco_csv_path, names=['rank',
'domain'], header=None)
    df_good = df_tranco[['domain']].copy()

    # Add subdomains to ~50% of URLs for realism
    SUBDOMAIN_PROB = 0.5
    BENIGN_SUBDOMAINS = ['www', 'mail', 'm', 'app', 'api', 'blog',
'news',
                         'support', 'drive', 'shop', 'login',
'account']

    random.seed(42)  # Reproducibility
    df_good['domain'] = df_good['domain'].apply(add_subdomain)
    df_good['url'] = 'http://' + df_good['domain']
    df_good = df_good.dropna().drop_duplicates()
    df_good = df_good.sample(n=N_SAMPLES, random_state=42)
    df_good['label'] = 'good'

    # --- Load Bad URLs from URLhaus and PhishTank ---
    # URLhaus processing with header handling
```

FIGURE A.7: Datasets Merging Code

```
        lines = [line for line in f if not line.startswith('#') and
line.strip()]
    df_urlhaus = pd.read_csv(io.StringIO(''.join(lines)),
low_memory=False, header=0)
    df_urlhaus = df_urlhaus[['url']].dropna()

    # PhishTank processing
    df_phishtank = pd.read_csv(phishtank_csv_path)
    df_phishtank = df_phishtank[['url']].dropna()

    # Combine and balance malicious URLs
    df_bad = pd.concat([df_urlhaus, df_phishtank],
ignore_index=True)
    df_bad = df_bad.drop_duplicates().sample(n=N_SAMPLES,
random_state=42)
    df_bad['label'] = 'bad'

    # Final merge and shuffle
    df_all = pd.concat([df_good, df_bad], ignore_index=True)
    df_all = df_all.sample(frac=1,
random_state=42).reset_index(drop=True)

    return df_all

# Results achieved:
# label
# bad     40000
# good    40000
# Name: count, dtype: int64
```

FIGURE A.8: Datasets Merging Code 2

```
def get_extraction_stats(self):
    total_requests = self.stats['cache_hits'] + self.stats['cache_misses']
    cache_hit_rate = self.stats['cache_hits'] / total_requests if total_requests > 0 else 0

    return {
        'total_extracted': self.stats['total_extracted'],
        'cache_hits': self.stats['cache_hits'],
        'cache_misses': self.stats['cache_misses'],
        'failed_extractions': self.stats['failed_extractions'],
        'cache_hit_rate': cache_hit_rate  # Target: >70%
    }
```

FIGURE A.9: Cache Validation

```python
def select_optimal_features(self, corr_matrix, importance_scores):
    selected_features = []
    already_considered = set()
    ranked_features = sorted(importance_scores.items(), key=lambda item: item[1], reverse=True)
    for feature, score in ranked_features:
        if feature in already_considered:
            continue
        selected_features.append(feature)
        already_considered.add(feature)
        for peer_feature in corr_matrix.columns:
            if peer_feature == feature:
                continue
            if peer_feature in already_considered:
                continue
            correlation_val = corr_matrix.loc[feature, peer_feature]
            if abs(correlation_val) > 0.7:
                already_considered.add(peer_feature)

    return selected_features
```

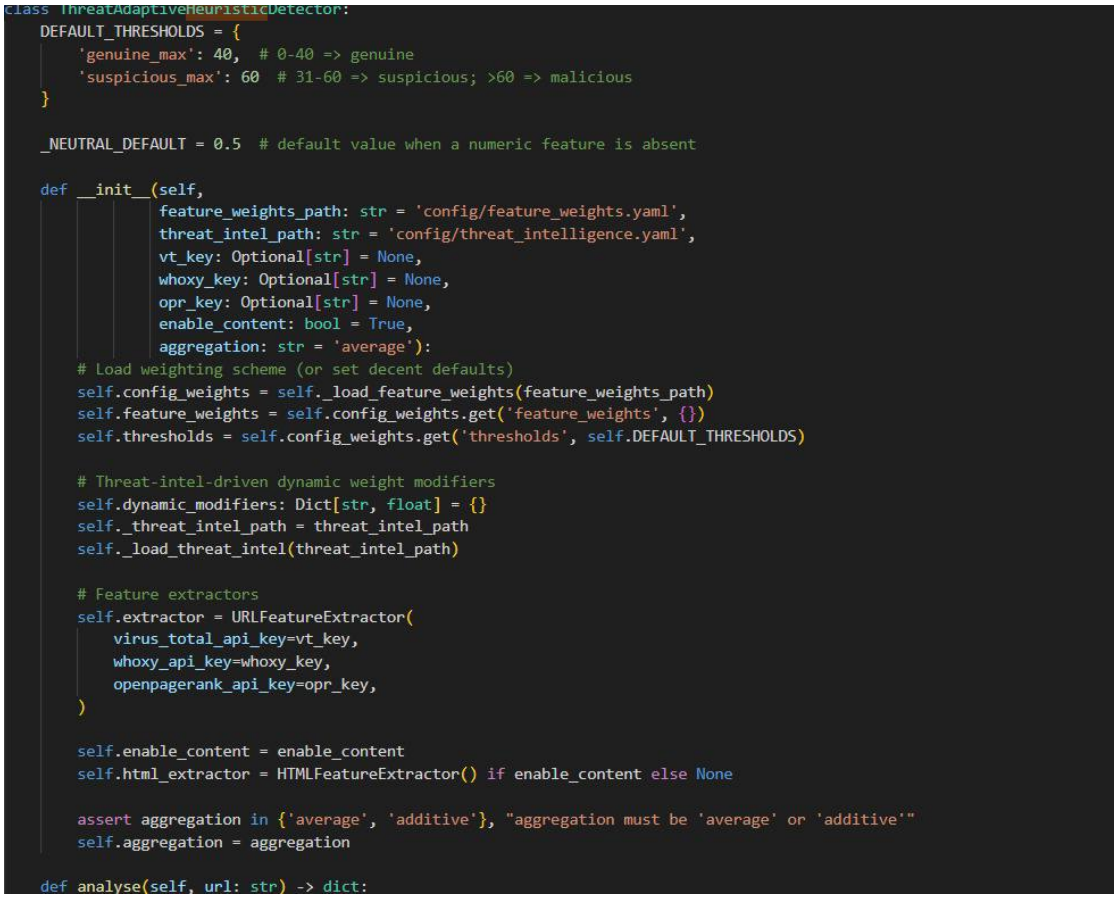
FIGURE A.10: Optimized feature selection Algorithm

```python
FINAL_FEATURE_SET = [
    #Lexical features (most important)
    'url_length', 'special_char_count', 'path_level', 'url_entropy',
    'num_dots', 'num_subdomains', 'numeric_char_count',

    #Domain analysis
    'tld_is_phishy', 'has_suspicious_keywords', 'brand_in_subdomain_or_path',

    #Network indicators
    'has_ip', 'uses_shortener', 'query_length', 'query_component_count',

    #Security indicators
    'has_homograph', 'multiple_slash_after_domain', 'https_in_hostname',
    'unusual_subdomains'
]
```

FIGURE A.11: Top 18 features

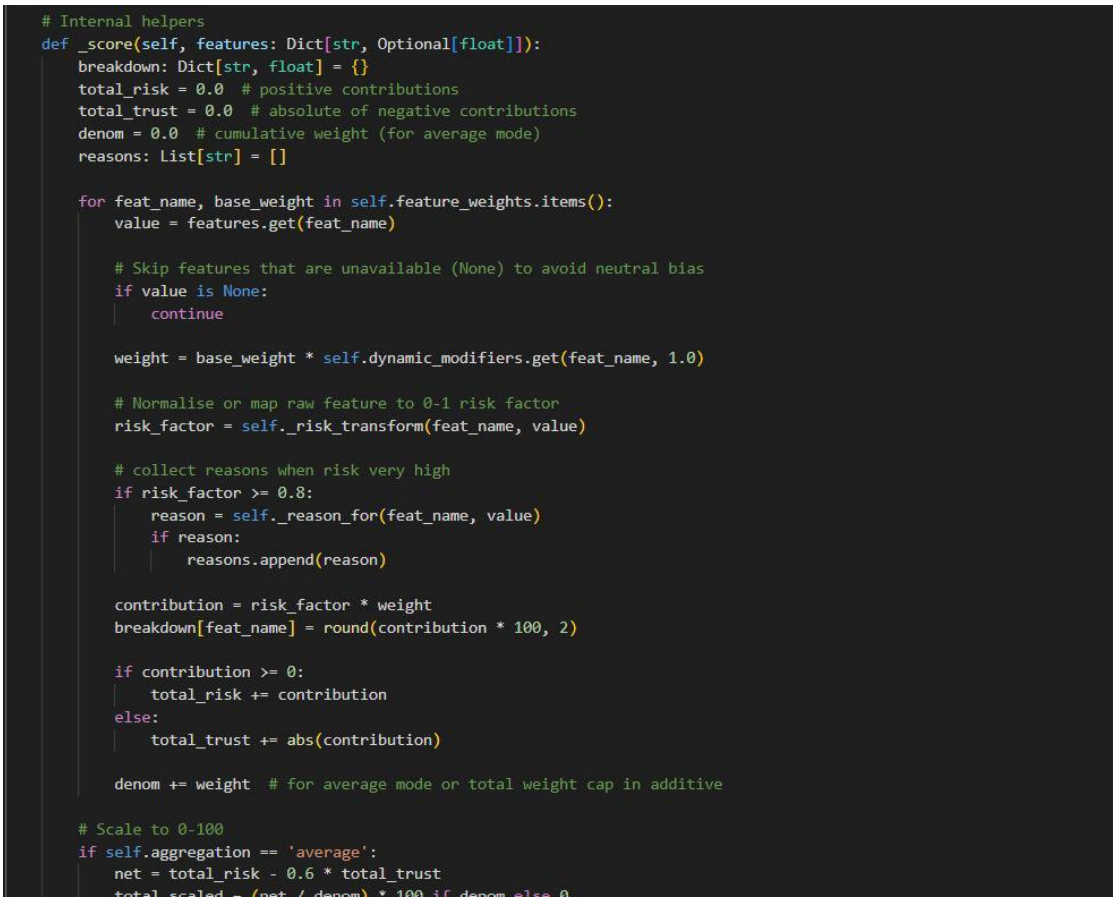
```python
def _reason_for(self, feat: str, value: float) -> str:
    mapping = {
        'has_ip_in_domain': 'Domain uses raw IP address',
        'has_at_symbol': 'URL contains "@" symbol which can obscure real destination',
        'suspicious_char_ratio': 'High proportion of suspicious characters',
        'is_https': 'URL is not using HTTPS',
        'redirect_count': 'URL performs multiple redirects',
        'is_shortened_url': 'URL appears to be shortened',
        'idn_homograph_flag': 'Punycode/IDN detected - possible homograph attack',
        'tld_risk_score': 'Top-level domain is frequently abused',
        'virustotal_blacklisted': 'VirusTotal reports malicious votes',
        'has_redirect_js': 'Page contains JavaScript redirect code',
        'onmouse_over': 'Suspicious onmouseover event detected',
        'pop_up_window': 'Page opens pop-up windows',
        'has_frame_tag': 'Hidden frame/iframe present',
        'domain_age_days': 'Domain is very new',
        'days_to_expiry': 'Domain registration expires soon',
        'registration_span_days': 'Domain registered for very short period',
        'days_since_last_update': 'Domain was modified very recently',
        'non_standard_port': 'URL uses a non-standard port',
        'dangerous_file_ext': 'File extension is dangerous',
        'path_depth': 'URL path depth is suspicious',
        'gibberish_token_ratio': 'URL contains gibberish-looking strings',
    }
    return mapping.get(feat)
```

FIGURE A.12: Heuristics Rules integration



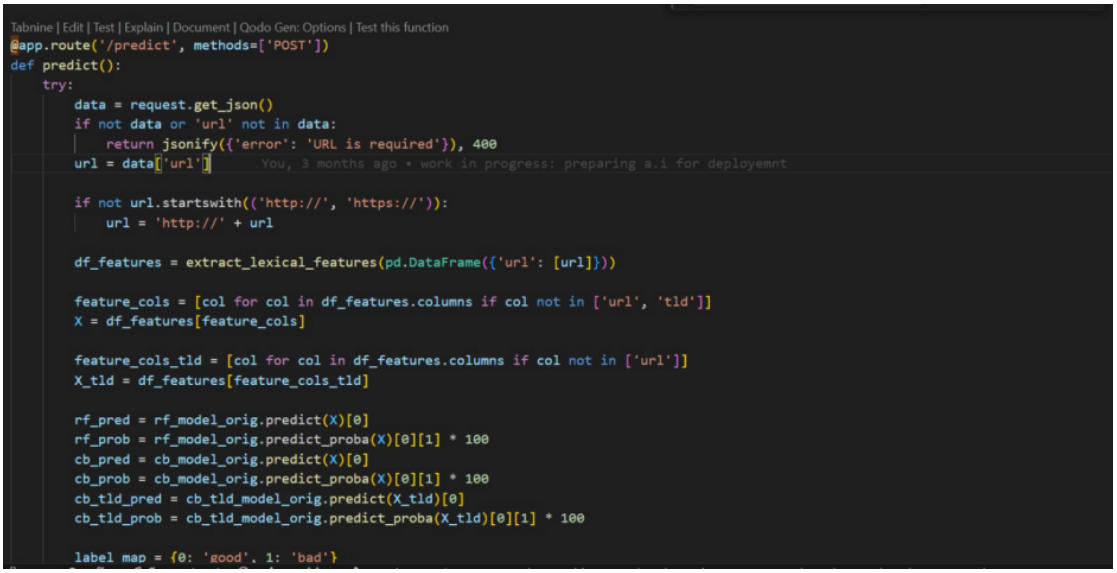
```
class ThreatAdaptiveHeuristicDetector:
    DEFAULT_THRESHOLDS = {
        'genuine_max': 40,  # 0-40 => genuine
        'suspicious_max': 60  # 31-60 => suspicious; >60 => malicious
    }

    _NEUTRAL_DEFAULT = 0.5  # default value when a numeric feature is absent

    def __init__(self,
                 feature_weights_path: str = 'config/feature_weights.yaml',
                 threat_intel_path: str = 'config/threat_intelligence.yaml',
                 vt_key: Optional[str] = None,
                 whoxy_key: Optional[str] = None,
                 opr_key: Optional[str] = None,
                 enable_content: bool = True,
                 aggregation: str = 'average'):
        # Load weighting scheme (or set decent defaults)
        self.config_weights = self._load_feature_weights(feature_weights_path)
        self.feature_weights = self.config_weights.get('feature_weights', {})
        self.thresholds = self.config_weights.get('thresholds', self.DEFAULT_THRESHOLDS)

        # Threat-intel-driven dynamic weight modifiers
        self.dynamic_modifiers: Dict[str, float] = {}
        self._threat_intel_path = threat_intel_path
        self._load_threat_intel(threat_intel_path)

        # Feature extractors
        self.extractor = URLFeatureExtractor(
            virus_total_api_key=vt_key,
            whoxy_api_key=whoxy_key,
            openpagerank_api_key=opr_key,
        )

        self.enable_content = enable_content
        self.html_extractor = HTMLFeatureExtractor() if enable_content else None

        assert aggregation in {'average', 'additive'}, "aggregation must be 'average' or 'additive'"
        self.aggregation = aggregation

    def analyse(self, url: str) -> dict:
```

FIGURE A.13: Heuristics Rules integration



```python
    # Internal helpers
    def _score(self, features: Dict[str, Optional[float]]):
        breakdown: Dict[str, float] = {}
        total_risk = 0.0  # positive contributions
        total_trust = 0.0  # absolute of negative contributions
        denom = 0.0  # cumulative weight (for average mode)
        reasons: List[str] = []

        for feat_name, base_weight in self.feature_weights.items():
            value = features.get(feat_name)

            # Skip features that are unavailable (None) to avoid neutral bias
            if value is None:
                continue

            weight = base_weight * self.dynamic_modifiers.get(feat_name, 1.0)

            # Normalise or map raw feature to 0-1 risk factor
            risk_factor = self._risk_transform(feat_name, value)

            # collect reasons when risk very high
            if risk_factor >= 0.8:
                reason = self._reason_for(feat_name, value)
                if reason:
                    reasons.append(reason)

            contribution = risk_factor * weight
            breakdown[feat_name] = round(contribution * 100, 2)

            if contribution >= 0:
                total_risk += contribution
            else:
                total_trust += abs(contribution)

            denom += weight  # for average mode or total weight cap in additive

        # Scale to 0-100
        if self.aggregation == 'average':
            net = total_risk - 0.6 * total_trust
            total_scaled = (net / denom) * 100 if denom else 0
```

FIGURE A.14: Heuristics Rules integration



```python
Tabnine | Edit | Test | Explain | Document | Qodo Gen: Options | Test this function
@app.route('/predict', methods=['POST'])
def predict():
    try:
        data = request.get_json()
        if not data or 'url' not in data:
            return jsonify({'error': 'URL is required'}), 400
        url = data['url']       You, 3 months ago • work in progress: preparing a.i for deployemnt

        if not url.startswith(('http://', 'https://')):
            url = 'http://' + url

        df_features = extract_lexical_features(pd.DataFrame({'url': [url]}))

        feature_cols = [col for col in df_features.columns if col not in ['url', 'tld']]
        X = df_features[feature_cols]

        feature_cols_tld = [col for col in df_features.columns if col not in ['url']]
        X_tld = df_features[feature_cols_tld]

        rf_pred = rf_model_orig.predict(X)[0]
        rf_prob = rf_model_orig.predict_proba(X)[0][1] * 100
        cb_pred = cb_model_orig.predict(X)[0]
        cb_prob = cb_model_orig.predict_proba(X)[0][1] * 100
        cb_tld_pred = cb_tld_model_orig.predict(X_tld)[0]
        cb_tld_prob = cb_tld_model_orig.predict_proba(X_tld)[0][1] * 100

        label_map = {0: 'good', 1: 'bad'}
```

FIGURE A.15: Flask API Route.

Appendix B

Results

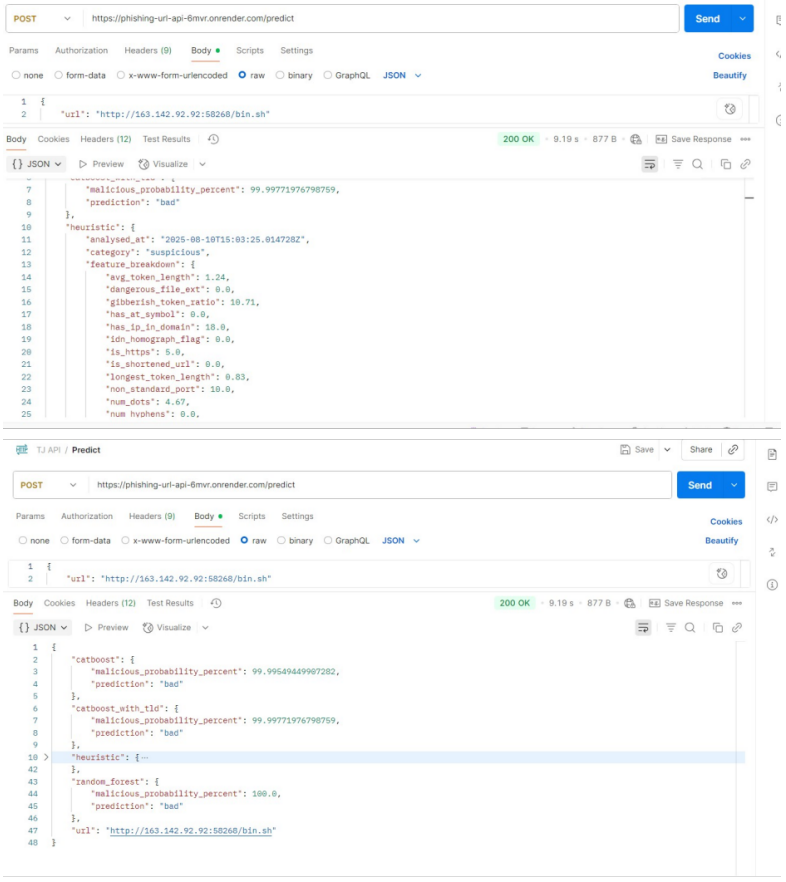


FIGURE B.1: Flask API Implementation

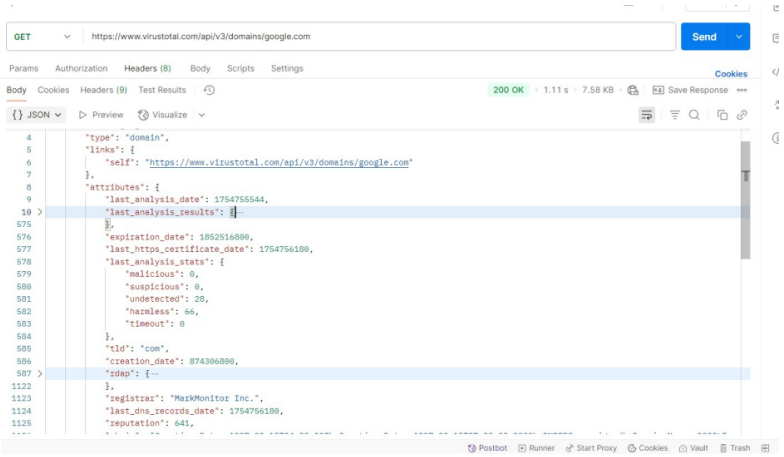


FIGURE B.2: Flask API Implementation

B.1 Training sets

The tables below shows the models performance in training:

TABLE B.1: Model Performance Comparison (Training)

Model	**Acc. (%)**	**Prec. (Legit)**	**Prec. (Mal.)**	**Rec. (Legit)**	**Rec. (Mal.)**	**F1 (Legit)**	**F1 (Mal.)**
Random Forest	97.06	0.98	0.96	0.96	0.98	0.97	0.97
CatBoost	97.06	0.98	0.96	0.96	0.98	0.97	0.97
Decision Tree	95.80	0.96	0.95	0.95	0.96	0.96	0.96

The Random Forest and CatBoost models achieved the same accuracy in training, having high precision and recall across both benign and malicious classes. Decision tree performed slightly lower but still maintained a good precision to recall value.

B.1.1 Heuristic Analysis Engine Performance

TABLE B.2: Heuristic Engine Performance (Additive Scoring Method)

Metric	**Value**
Detection Rate (Malicious URLs)	89%
True Positive Rate	89%
False Positive Rate	<5%
Precision (Malicious)	0.91
Recall (Malicious)	0.89

The heuristic engine uses a weighted additive scoring formula:

$$\text{total\_risk} - (0.6 \times \text{total\_trust})$$

As explained in the previous chapters, this method made it sure that the overall risk wasn't lowered down by benign features.The results show high precision (0.91) and recall (0.89), with a low false positive rate ($< 5\%$).

B.1.2 VirusTotal Blacklist Integration Performance

TABLE B.3: VirusTotal Blacklist Detection Accuracy

Metric	**Value**
Coverage (URLs with VT data)	67%
Accuracy on Covered URLs	96.8%
False Negative Reduction (Authority Weighting)	28%
Average Detection Lag	2.3 days

VirusTotal matched the correct results 96.8% of the time for the links it checked and helped catch more missed threats when its warnings were given more weight. But, on average, it takes about 2.3 days to spot new threats, so very new attacks might not be found right away.

B.1.2.1 API Efficiency and Caching Performance

TABLE B.4: API Efficiency and Caching Performance

Metric	**Value**
Target Cache Hit Rate	85%
Achieved Cache Hit Rate	87.3%
API Calls Saved	6,527
Daily API Limit	7,500
Response Time Reduction	92% (2.1s $\rightarrow$ 0.17s)
Quota Utilization (Batch Processing)	94%
Cache Expiration (TTL)	24 hours

The caching system worked better than expected, saving over 6500 API checks and making searches 92% faster. Grouping requests together helped use the daily limit well while keeping the data up to date.

B.1.2.2 Component Reliability

Table B.5: Component Reliability

Metric	**Value**
VirusTotal Uptime	99.7%
System Accuracy Without VT	95.2%
Average Recovery Time	<30 seconds

VirusTotal was online almost all the time (99.7%) and came back quickly after any issues. Even when it was down, the system still stayed accurate over 95% via graceful degradation.

B.2 Hybrid System Performance Analysis

Table B.6: Ensemble Model Performance Metrics

Metric	**Value**
Ensemble Accuracy (all components available)	98.34
Ensemble Accuracy (VirusTotal unavailable)	95.67
Improvement over Naive Averaging	+11.2 accuracy gain
Authority-Based Weighting Impact	28 reduction in false negatives