WWW.IJRPR.COM

International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

PhishGuard AI: A Machine Learning-Based Framework for Phishing URL Detection and Risk Classification

Mrs. Sabeetha M[1], Harshitha B M[2], Hemalatha M[3], Henry Joshua J[4], Indhu M5

1 Assistant Professor, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, India
2 Student, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, India.
3 Student, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, India.
4 Student, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, India.
5 Student, Department of Computer Science and Engineering, Sri Shakthi Institute of Engineering and Technology, India.

ABSTRACT :

Phishing websites imitate legitimate online services to trick users into revealing credentials, financial details and other sensitive information. PhishGuard AI is a web-based phishing URL detection system that combines a React frontend, a Flask backend and a Gradient Boosting classifier. The system extracts 27 URL and URL and hostname-derived features from a submitted URL and uses the trained model to classify the URL as legitimate or potentially phishing. The reported dataset contains 20,000 labelled URLs, including 10,007 legitimate and 9,993 phishing examples. In the recorded held-out evaluation, the model achieved 83.83% accuracy. For the phishing class, precision was 0.88, recall was 0.78 and F1-score was 0.83. These results indicate useful preliminary classification performance, but false positives and false negatives remain. A configured trusted-host list may provide an additional recognition check; it does not guarantee that a site is safe. The system is intended as a first-line warning aid rather than a replacement for browser security controls.

Keywords: PhishGuard AI, Phishing URL Detection, Machine Learning, Gradient Boosting, Hostname Features, React, Flask, Cybersecurity.

I. INTRODUCTION

The use of websites for communication, education, shopping and financial services has made online security increasingly important. Phishing is a common cyber threat in which attackers create deceptive websites or links that appear to belong to trusted organizations. Users who visit these pages may disclose passwords, banking information or other confidential data. Effective detection is challenging because phishing techniques and domain patterns change over time.

Traditional protection methods may depend on blocklists, signatures or users recognizing suspicious signs. These methods can identify known threats, but newly created or modified phishing websites may not immediately appear in existing lists. Analysing URL and hostname characteristics provides another signal that can support users before they interact with a suspicious link.

This paper presents PhishGuard AI, a phishing URL detection application that combines a React-based frontend, a Flask backend and a Gradient Boosting machine-learning model. The model uses 27 URL and URL and hostname-derived features extracted from a submitted URL. The dataset used for the reported experiment contains 20,000 labelled records: 10,007 legitimate URLs and 9,993 phishing URLs. The application also includes a configured trusted-host list as a supplementary check. The system's output is a preliminary risk classification, not proof that a website is safe.

1. To build a web interface through which users can submit a URL for phishing-risk analysis.
2. To extract a consistent set of 27 URL and URL and hostname-derived features from a submitted URL.
3. To connect the frontend to a Flask backend that performs feature preparation and model inference.
4. To use a trained Gradient Boosting classifier to return a readable prediction to the user.
5. To provide a modular foundation for future improvements such as broader URL features, model evaluation and updated threat intelligence.

The system is intended to support awareness and preliminary screening rather than guarantee that a website is safe. The reported evaluation achieved 83.83% accuracy on a held-out test set; therefore, some legitimate URLs may be flagged and some phishing URLs may be missed. Users should continue to verify website addresses and follow safe browsing practices.

The remainder of this paper reviews related work, describes the proposed methodology, explains the implementation, presents the recorded evaluation results and discusses limitations and future enhancements.

II. LITERATURE SURVEY

Phishing detection has been investigated through blacklists, heuristic rules, visual similarity analysis and machine-learning classifiers. These approaches use different signals and have different strengths and limitations. PhishGuard AI focuses on URL and hostname-derived features and supervised classification within a lightweight web application.

Blacklist-Based Detection. A blacklist compares a submitted URL or domain against previously identified malicious addresses. It can be effective for known threats, but its coverage depends on how quickly lists are updated and may not identify newly registered phishing domains.

Heuristic and URL-Based Analysis. Heuristic systems inspect characteristics such as URL length, unusual symbols, subdomain structure and domain-related patterns. These indicators can be useful but may also occur in legitimate URLs, so individual rules are not sufficient for a reliable decision.

Machine-Learning Approaches. Supervised learning algorithms can learn patterns from labelled examples of legitimate and phishing URLs. Their effectiveness depends on feature quality, dataset coverage, class balance and evaluation methodology. Models can also degrade when attacker behaviour changes.

Tree-Based Classification. Gradient Boosting combines multiple weak decision trees in a sequential learning process. It can model non-linear relationships among structured features and is suitable for tabular classification tasks, but it requires appropriate training and validation to avoid overfitting.

Domain and Hostname Signals. Hostname-related features can capture patterns in domain length, subdomain composition and other syntactic properties. Such signals are relatively lightweight to compute, although hostname-only analysis may miss malicious paths, page content, redirects or visual impersonation.

Web-Based Security Tools. A client–server design separates the user interface from the detection logic. In PhishGuard AI, the React frontend collects the URL and displays the result, while the Flask backend prepares features and invokes the trained classifier.

Research Gap. No single URL feature or classifier can identify every phishing website. PhishGuard AI demonstrates an integrated workflow for URL/hostname-feature extraction and machine-learning prediction; broader evaluation and additional signals are required before making claims about real-world detection effectiveness.

III. PROPOSED METHODOLOGY

PhishGuard AI follows a client–server architecture. The user submits a URL through the React frontend, the Flask backend validates and processes the input, a feature-extraction component derives 27 hostname-based values, and the trained Gradient Boosting model returns a classification for display. The recorded model evaluation is reported separately from the application's functional workflow.

System Architecture. The presentation layer is the React interface. The application layer is a Flask API that receives the URL, validates the request and coordinates feature extraction and prediction. The model layer contains the trained Gradient Boosting classifier and its expected feature order. A trusted-host list can be consulted as a supplementary check. The components must use consistent input validation and feature definitions.

Workflow. The user enters a URL and submits it for analysis. The frontend sends the input to the Flask backend. The backend checks that the input is present and has an acceptable URL format, extracts the hostname-based features in the order expected by the model, and performs inference. The result is returned to the frontend and displayed as a prediction. Errors such as invalid input or unavailable model services should produce a clear response.

URL Input and Validation. The application accepts a URL supplied by the user and checks that it is syntactically usable before feature extraction. Input validation reduces malformed requests but does not itself establish whether a website is safe. The application should avoid automatically opening user-submitted URLs unless that behaviour is explicitly designed and secured.

Hostname Feature Extraction. The feature-extraction stage derives the project's 27 URL and URL and hostname-derived features. The same definitions, ordering, types and preprocessing used during model training must be applied during prediction. Any mismatch between training-time and inference-time features can cause incorrect results or runtime errors.

Machine-Learning Prediction. The extracted feature vector is passed to the trained Gradient Boosting classifier. The model produces a class prediction that the backend converts into a clear response for the frontend. The prediction should be presented as an estimate rather than a guarantee, and confidence scores should be displayed only if they are meaningfully calibrated and supported by the implementation.

The dataset used in this experiment contains 20,000 labelled URL records with the columns 'url' and 'label'. The original dataset source and access date must be stated in the final manuscript using the actual source used by the project; these details are not available in the current project record and must not be guessed. The training script removes duplicate records before performing an 80:20 stratified train/test split with random_state=42. The reported test set contains 4,000 examples.

Trusted-Host Check and Security. A configured trusted-host list may be used as an additional recognition signal, but it must not be treated as a complete security boundary. Domain matching should be exact and carefully normalized to avoid misleading substring matches. API inputs should be validated, errors handled safely and model or dataset files protected from unauthorized modification.

IV. SYSTEM IMPLEMENTATION

The implementation is organized into a frontend, backend, feature-extraction component and prediction component.

The frontend is implemented with React and JavaScript. The backend uses Python with Flask to expose the prediction workflow through an API. The detection logic uses a trained Gradient Boosting classifier and a consistent hostname-feature extraction process. The recorded evaluation compares Gradient Boosting with a Random Forest baseline using cross-validation. Exact dependency versions and deployment configuration should be recorded in the project documentation.

Frontend Module. The React interface provides a field for entering a URL, a submit action, and an area for displaying the returned classification. The interface should show input or server errors clearly and avoid presenting the result as a guarantee of safety.

Backend API Module. The Flask service receives the frontend request, validates the submitted value, invokes feature extraction and model inference, and returns a structured response. The API should reject empty or malformed input and handle missing model assets or unexpected exceptions without exposing sensitive internal details.

Feature Extraction Module. The feature extractor parses the submitted URL and derives the 27 hostname-based features used by the model. Feature names, ordering, data types and preprocessing must remain consistent with the training pipeline.

Prediction Module. The prediction component loads the trained Gradient Boosting classifier and applies it to the prepared feature vector. The backend maps the output to the application's supported result labels and returns the prediction to the frontend.
Frontend–Backend Integration. The React application sends the URL to the Flask API using an HTTP request. The frontend handles the response and displays the result. During deployment, the API address, CORS configuration and request/response schema must be configured consistently.
Model and Dataset Handling. The saved model should be loaded from a controlled location and match the feature-extraction code. Training, validation and test data should be kept separate, and test examples should not influence feature selection or model tuning.
Error Handling. The application should handle invalid URLs, missing request fields, feature-extraction failures, unavailable model files, unexpected prediction errors and network failures. Error messages should help users recover without exposing stack traces or server configuration.
Functional testing should cover URL submission, input validation, feature generation, model inference, response formatting and error handling. Only tests actually executed against the final code should be reported as passed. The following items are recommended test cases and must not be described as completed unless test output is available:
Submit a valid URL: the frontend sends the request and receives a structured response.
Submit an empty or malformed URL: the backend rejects the input with a suitable message.
Submit a URL with a valid hostname: the feature extractor returns the expected 27 features.
Run model inference: the backend returns a supported prediction label without a feature-order error.
Submit a hostname with unusual characters or subdomains: parsing and validation behave consistently.
Test a domain present in the configured trusted-host list: the supplementary check follows the documented matching rules.
Stop or remove the model asset: the API reports a controlled service error rather than crashing.
Send a request with missing or incorrectly typed fields: the backend rejects the request safely.
Interrupt the network connection: the frontend reports that the result could not be retrieved.

V. ADVANTAGES

1. Quick Preliminary Screening. Users can submit a URL and receive a machine-learning prediction through a single interface.
2. Lightweight Feature Analysis. Hostname-based features can be extracted without requiring a full browser rendering of the target page.
3. Integrated User Experience. The React frontend and Flask backend provide a clear separation between interaction and prediction logic.
4. Consistent Classification Workflow. Applying the same feature definitions during inference helps make model input processing repeatable.
5. Extensible Architecture. The frontend, API, feature extractor and classifier can be maintained as separate components.
6. Supplementary Trusted-Host Recognition. A trusted-host list can help recognize configured domains when matching is implemented carefully.
7. Support for Security Awareness. A clear prediction can encourage users to pause and inspect suspicious links before interacting with them.
8. Potential for Further Evaluation. The project can be extended with reproducible model testing, additional URL signals and updated datasets. These advantages depend on correct implementation and validation; the tool should not be treated as a replacement for browser security controls.

VI. RESULTS AND ANALYSIS

The recorded held-out evaluation of the Gradient Boosting classifier achieved 83.83% accuracy on 4,000 examples. The phishing-class precision was 0.88, recall was 0.78 and F1-score was 0.83. The confusion matrix was TN = 1,795, FP = 206, FN = 441 and TP = 1,558. These values indicate that the classifier detected many phishing URLs but missed 441 phishing examples in this test set. Results should be reproduced from the final training script before publication, because metrics may change if the dataset, split or preprocessing differs.
A. Implementation Status. The project is designed with a React frontend, a Flask backend, a 27-feature URL and hostname extraction process and a Gradient Boosting prediction component. The backend health endpoint was previously observed returning a status response. Final submission evidence should include the exact test output and screenshots or logs from the same code version used for the paper.
B. Functional Analysis. Functional tests should verify URL submission, input validation, feature generation, model inference, response formatting and error handling. Record the input condition, expected outcome, actual outcome and pass/fail status for each test. Do not claim a total number of passing tests unless the final test-run report confirms it.
C. Model Evaluation. In the recorded held-out evaluation, the model achieved 83.83% accuracy. For legitimate URLs, precision was 0.80, recall was 0.90 and F1-score was 0.85. For phishing URLs, precision was 0.88, recall was 0.78 and F1-score was 0.83. The macro-average precision, recall and F1-score were each approximately 0.84. The confusion matrix contained 1,795 true negatives, 206 false positives, 441 false negatives and 1,558 true positives. Re-run the evaluation and confirm these numbers against the final code before submission.
D. Performance Analysis. Classification metrics are available from the recorded model evaluation, but a verified end-to-end response-time benchmark is not included here. Measure repeated frontend-to-backend requests and report the number of samples, median latency and a percentile such as the 95th percentile before making speed claims.
E. Limitations. (i) Hostname-only features may not detect phishing that depends on URL paths, page content or visual similarity. (ii) Attack patterns and domain-registration practices change over time. (iii) Trusted-host lists can become outdated. (iv) Model predictions can be wrong, and an apparently legitimate result must not be interpreted as proof that a site is safe.
The training script compares Random Forest and Gradient Boosting using five-fold StratifiedKFold cross-validation (shuffle=True, random_state=42) and selects the model with the higher mean F1-score. In the recorded run, Random Forest achieved a mean CV F1-score of 0.8235 ± 0.0060 and Gradient Boosting achieved 0.8363 ± 0.0063. The held-out evaluation used an 80:20 stratified train/test split with random_state=42 (4,000 test examples). The final test results must remain separate from model selection.
Classification performance: the recorded held-out accuracy was 83.83%; phishing precision was 0.88, recall was 0.78 and F1-score was 0.83.

Error analysis: the recorded confusion matrix was TN = 1,795, FP = 206, FN = 441 and TP = 1,558.
Feature pipeline: verify that all 27 expected features are extracted in the same order and data types used during training.
Generalization: the reported results are from a held-out split of the supplied dataset; evaluation on a newer or external dataset is still recommended.
API performance: median and percentile response time for representative requests.
Reliability: correct handling of invalid input, unavailable model files and network failures.
Security: validation of API inputs, safe error responses and protection of model and configuration assets.
Usability: feedback from representative users about clarity of predictions and warnings.
G. Discussion. The recorded results show that Gradient Boosting can distinguish many legitimate and phishing URLs using hostname-related features, with 83.83% held-out accuracy. However, the phishing recall of 0.78 means that some phishing URLs were missed, while the false-positive count shows that some legitimate URLs were incorrectly flagged. The model should therefore be used as a preliminary warning mechanism. The reported results apply to the evaluated dataset and do not establish performance against all current phishing websites.

VII. CONCLUSION

This paper presented PhishGuard AI, a web-based phishing URL detection project combining a React frontend, a Flask backend, hostname-based feature extraction and a Gradient Boosting classifier. The model analyses 27 URL and URL and hostname-derived features. On the recorded held-out evaluation of 4,000 examples, it achieved 83.83% accuracy; phishing precision, recall and F1-score were 0.88, 0.78 and 0.83 respectively. The confusion matrix included 441 false negatives and 206 false positives, showing that errors remain. The application should be treated as a preliminary screening aid, not a guarantee that a URL is safe. The reported metrics must be reproduced from the final code and dataset before journal submission.
Future work should evaluate the model on recent and independent URL datasets, document the dataset source and preprocessing, compare additional classifiers, measure API response time, and strengthen input validation and trusted-host handling. Combining hostname features with carefully collected URL-path, redirect or domain-reputation signals may improve coverage, but each extension requires separate evaluation.

VIII. FUTURE ENHANCEMENTS

Expanded URL Features. Future versions can examine additional URL components, redirect behaviour and domain metadata where these signals can be collected safely and lawfully.
Model Evaluation and Comparison. The Gradient Boosting model can be compared with suitable baseline classifiers using the same train/test split and metrics, with careful prevention of data leakage.
Dataset Updates. Training and evaluation data can be refreshed with labelled, recent examples and documented sources to better reflect changing phishing patterns.
Explainable Predictions. The interface may show understandable indicators that contributed to a prediction, provided that explanations accurately reflect the trained model and do not expose sensitive implementation details.
Threat-Intelligence Integration. Carefully validated feeds or domain-reputation services could provide an additional signal, with clear handling for stale, unavailable or conflicting data.
Improved Trusted-Host Management. The trusted-host list can be maintained through controlled updates and exact domain matching, with safeguards against unsafe or misleading entries.
Deployment and Monitoring. The service can be deployed with appropriate logging, health checks, dependency management and monitoring while avoiding unnecessary storage of submitted URLs or sensitive data.
Security and Privacy. Further work should review API security, access restrictions, safe error handling, retention policies and protection of model and configuration files.
Automated Testing and Scalability. Automated tests can cover URL validation, feature extraction, API responses and prediction behaviour, followed by load testing to identify performance bottlenecks.

IX. REFERENCES :

1. Anti-Phishing Working Group (APWG). Phishing Activity Trends Reports. https://apwg.org/trendsreports/
2. National Institute of Standards and Technology (NIST). Artificial Intelligence Risk Management Framework (AI RMF 1.0). 2023. https://doi.org/10.6028/NIST.AI.100-1
3. Sahingoz, O. K., Buber, E., Demir, O., & Diri, B. (2019). Machine learning based phishing detection from URLs. Expert Systems with Applications, 117, 345–357. https://doi.org/10.1016/j.eswa.2018.09.029
4. Ma, J., Saul, L. K., Savage, S., & Voelker, G. M. (2009). Beyond blacklists: learning to detect malicious web sites from suspicious URLs. Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 1245–1254. https://doi.org/10.1145/1557019.1557153
5. Dataset source: replace this entry with the exact dataset title, author/organization, repository URL, and access date for the dataset actually used in this project. Do not cite a dataset source unless it matches the downloaded file.
6. Scikit-learn Developers. Gradient Boosting Classifier documentation. https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html
7. Flask Documentation. https://flask.palletsprojects.com/
8. React Documentation. https://react.dev/

9. OWASP Foundation. OWASP Top 10:2021. https://owasp.org/Top10/
10. Open Worldwide Application Security Project (OWASP). Input Validation Cheat Sheet. https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html
11. Mozilla. HTTP Overview. https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview
12. Google Safe Browsing. Safe Browsing APIs. https://developers.google.com/safe-browsing
13. MITRE. CWE: Common Weakness Enumeration. https://cwe.mitre.org/
14. Python Software Foundation. Python Documentation. https://docs.python.org/3/
15. Gradient Boosting: Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. Annals of Statistics, 29(5), 1189–1232. https://doi.org/10.1214/aos/1013203451
16. Werkzeug Documentation. https://werkzeug.palletsprojects.com/
17. MDN Web Docs. URL API. https://developer.mozilla.org/en-US/docs/Web/API/URL