AI-Based Fake QR Code Detection with a QR Risk Score Using Machine Learning

Dr. Mrs. Jaya M. Bhattad[1], Kalyani Pradip Derkar[2],Gopal Premkumar Deotare[3],Harshal Harishchandra Chopkar[4]

jaya.bhattad@pcenagpur.edu.in, kalyaniderkar20@gmail.com, gopaldeotare16@gmail.com, harshalchopkar7112004@gmail.com

Department of Electronics and Telecommunication Engineering
Priyadarshini College of Engineering, Nagpur, Maharashtra, India – 440019

Abstract : A QR code is just a printed pattern, so a person scanning one has no way to tell where it will lead, and QR codes now carry payments, web links and login prompts. Quishing, meaning phishing delivered through QR codes, takes advantage of this by steering people to fake pages, executable downloads or altered UPI payment requests. We built a prototype that examines what a QR code actually contains instead of simply decoding it. An uploaded image passes through several OpenCV preprocessing steps, with ZBar and ZXing-C++ available as optional backup decoders. From the recovered payload, 23 security features are computed in six groups: payload metrics, protocol and host, lexical counts, domain characteristics including Shannon entropy, malware and phishing indicators, and UPI parameters. A Random Forest classifier with 450 trees, a maximum depth of 12 and balanced class weights then estimates how likely the payload is to be malicious. That probability is blended with rule-based penalties to give a QR Risk Score between 0 and 100, which maps to Safe, Medium Risk or High Risk and appears alongside plain-language warnings. Every scan is stored in SQLite and can be reviewed in a Streamlit dashboard. We evaluated on a stratified 80:20 split of a labelled dataset of 450 payloads (295 safe, 155 malicious); the classifier reached 98.89% test accuracy, making one error among 90 test payloads, and five-fold cross-validation gave 100%. These figures should be read as optimistic. In this dataset every safe URL uses HTTPS and every malicious URL uses HTTP, many payloads follow repeated templates, and the two classes remain perfectly separable even after the protocol features or the repeated rows are removed. A supplementary test on 80,000 URLs from a larger collection supports this caution: the same model scored 86.3% accuracy and missed about 27% of the malicious URLs. The results show that the pipeline works from end to end but say little about real-world performance, and larger, more realistic datasets with independent testing are needed before stronger claims can be made.

IndexTerms - QR code security, quishing, phishing detection, Random Forest, QR Risk Score, UPI security, explainable warnings, payload analysis.

I. INTRODUCTION

Quick Response (QR) codes began as labels for tracking parts on factory floors and have since settled into daily life. In India they are most familiar from Unified Payments Interface (UPI) payments, yet they also open websites, place restaurant orders, hold tickets and share Wi-Fi credentials. One scan and no typing is the appeal. It is also the weak point: a QR code is just a grid of dark and light squares, and looking at it tells nobody whether it leads to a bank's payment page or to an attacker's login form.

Researchers have been looking at this problem for a while. Kharraz et al. studied malicious QR codes found in the wild and showed how they send users to harmful destinations [1]. The practice is now known as quishing. A sticker placed over a genuine code at a shop counter, a flyer that points to a fake prize page, or a code that triggers an .apk download all rely on the victim trusting the image. With payments, an attacker can also tamper with a UPI parameter such as the payee address, so the money lands in another account while the rest of the transaction looks ordinary.

Most scanners decode the content and then open it, or ask whether to open it. They do not judge whether the decoded text looks suspicious, and they rarely say why a code might be dangerous. Our prototype targets that gap. After decoding, a Random Forest classifier converts the properties of the payload into a QR Risk Score and a short list of reasons. The contributions are a multi-pass decoding stage, 23 security features that include UPI-specific ones, a score that combines the classifier with rule-based penalties, and a Streamlit dashboard with SQLite logging. How far the evaluation can be trusted is limited by the size of the dataset, as Sections XI to XIII explain.

II. LITERATURE REVIEW

Ozuberk and Cinar compared machine-learning and deep-learning models for classifying malicious QR codes and included explainable AI in the comparison [2]. In their reported results, DenseNet121 reached about 89.9% accuracy with a ROC-AUC of 0.94. They frame malicious QR detection as a question of choosing the right model, and their work shows that explainability is already part of the discussion. We rely on their published summary and did not repeat the study.

Ahamed et al. come at quishing from the supply-chain side [3]. They connect QR phishing risks to traceability in food supply chains and suggest blockchain and smart contracts for verifying QR codes, with the aim of improving security, traceability and trust. Verification here comes from infrastructure: it works when products and parties are registered in a shared ledger, which is quite different from someone scanning an unknown code in a public place.

Rafsanjani et al. built QsecR, a secure QR scanner based on a malicious URL detection framework [4]. By the reported summary, it uses 39 URL features and 4000 URLs and achieves about 93.50% accuracy and 93.80% precision. Of the three studies it is the nearest to ours, since it too judges the link behind a code from features of the decoded content. The underlying idea is older: Ma et al. showed that lexical and host-based properties of a URL can separate malicious sites from benign ones without loading the page [5]. We apply this principle to decoded QR payloads and extend it to UPI parameters.

The figures in [2]–[4] come from those authors' own data, so they cannot be compared with one another or with our results, and we ran no comparative experiment. Going by the reported summaries, none of these tools offers a graded risk score, a scan

log and UPI-specific features together, and that is where our prototype sits.

III. PROBLEM STATEMENT

A QR image does not show whether its decoded content is safe, and common scanners decode it without assessing the URL or payment data inside. The task is therefore to take a QR image, recover its payload, estimate how likely the payload is to be malicious or suspicious, and explain the estimate to the user in simple terms, all with a low-cost tool that runs on ordinary hardware.

IV. OBJECTIVES

The system was designed to decode QR images with a multi-stage OpenCV module backed by optional ZBar and ZXing-C++ fallbacks; to compute 23 security features covering payload, protocol, lexical, domain, malware and UPI aspects; to train a Random Forest that outputs a threat probability; to turn that probability into a 0–100 QR Risk Score with three levels and readable warnings; and to record every scan in SQLite, with a Streamlit dashboard for review.

V. PROPOSED METHODOLOGY

The method follows the order in which a scan is processed. The input is a QR image uploaded through the dashboard (JPG, JPEG or PNG), a camera capture or a test payload. Photographs and screenshots vary in size, contrast and sharpness, so the decoder works in passes: the original image, a grayscale copy, a contrast-enhanced copy (CLAHE), Otsu thresholding and adaptive Gaussian thresholding, plus a two-times bicubic enlargement for small codes and a recovery step for branded or payment QR codes. OpenCV's QR detector is tried on these variants first. If none of them yields a payload, ZBar and ZXing-C++ act as optional fallback decoders when they are installed.

The recovered string is classed as a UPI payment URI, a URL or plain text. URLs are parsed into scheme, host, path and query, and the host is split into domain and subdomain parts. From these, 23 numeric features are calculated, including the Shannon entropy [6] of the domain name. The feature vector is passed to a Random Forest classifier [7], which returns the probability that the payload is malicious. This probability is merged with rule-based penalties into a score from 0 to 100, mapped to a risk level and displayed with warnings, and the scan is saved in SQLite.

VI. DATASET

The prototype dataset begins with a CSV file of 450 labelled payloads (data/raw_data.csv), which the script prepare_dataset.py converts into the 23-feature table data/qr_dataset.csv. Class 0 (safe) contains 295 payloads: 265 HTTPS URLs of well-known sites, 20 well-formed UPI payment URIs and 10 plain-text strings such as table numbers, visitor passes and a Wi-Fi note. Class 1 (malicious or suspicious) contains 155 payloads: 125 HTTP URLs, among them 30 raw-IP hosts, suspicious TLDs, phishing-style and fake prize pages, and links to executable or script downloads such as .zip, .vbs, .scr or .apk files; 25 UPI URIs with a missing or fake payee address; and 5 plain-text scam messages. How the examples were originally gathered was not documented for this paper.

Since the data are text payloads, the classifier does not depend on image quality. Each payload is converted to the 23 features described in Section VII and carries a binary label. We split the data 80:20 with stratified sampling and a fixed random seed of 42, which gave 360 payloads for training and 90 for testing (59 safe and 31 malicious). No separate validation set was used, and five-fold stratified cross-validation served as an additional check.

Two properties of the dataset affect how the results should be read. First, it is heavily templated: only 410 of the 450 payload strings are unique, the safe URLs come from just 52 hosts and reuse a handful of paths such as /home, /help/faq and /products, and 75 rows repeat the feature vector of an earlier row. Consequently, 25 of the 90 test payloads have an identical feature vector in the training set. Second, the protocol separates the two classes: every safe URL uses HTTPS and every malicious URL uses HTTP, so a classifier can score well by looking at the protocol alone.

VII. FEATURE EXTRACTION AND FEATURE ENGINEERING

Table 1 lists the 23 features. Each one captures a property that attackers often exploit or that tends to differ between ordinary and suspicious links. No single feature proves bad intent, and the classifier weighs them together. We did not apply a separate feature-selection step.

Table 1. Security features used by the classifier (23 in total)

Category	Features
Payload metrics (4)	Data_Length, QR_Type_UPI, QR_Type_URL, QR_Type_TEXT
Protocol and host (3)	HTTPS, HTTP, Has_IP_Address
Lexical (5)	Number_of_Dots, Number_of_Slashes, Number_of_Hyphens, Number_of_Digits, Number_of_Special_Characters
Domain intelligence (5)	Domain_Length, Domain_Entropy, Number_of_Subdomains, Is_URL_Shortener, Suspicious_TLD
Malware and phishing (3)	Has_Executable_Ext, Suspicious_Keyword_Count, Has_Suspicious_Keyword
UPI / payment (3)	UPI_Payment_Address, UPI_Payee_Name, UPI_Merchant_Code

A. Payload, protocol and lexical features

Payload length and a one-hot code for its type let the model treat UPI strings, URLs and plain text differently; for payloads that are not URLs, the host-based features take default values. HTTPS encrypts the connection but says nothing about who runs the server, and phishing sites use it often, so in the rule layer HTTP only triggers a warning and carries no penalty. The trained forest behaves differently: HTTPS and HTTP are its two most influential features, which follows from how the dataset was built (Section VI) and is not a sound security signal. A raw IP address in place of a domain name is unusual for public services and often points to temporary hosting, although IP hosts also turn up in local test setups. Counts of dots, slashes, hyphens, digits and special characters describe the shape of the string, because long paths, many hyphens and deep nesting are common in links crafted to imitate a real address.

B. Domain intelligence

We measure how random a domain name looks with Shannon entropy [6]. Writing p(x) for the relative frequency of character x in the domain,

$$H(X) = -\Sigma_x\, p(x) \cdot \log_2 p(x) \qquad (1)$$

A higher value means the characters are spread more evenly, which is typical of machine-generated names. It is only an indicator, because some genuine brands have unusual names. The number of subdomains, the domain length and a lookup

against a list of known URL shorteners complete the group. Suspicious_TLD flags endings such as .xyz, .top or .click, which are cheap to register and common in abusive campaigns, although plenty of harmless sites use them as well.

C. Malware, phishing and UPI indicators

Has_Executable_Ext checks whether the link points to a file such as .exe or .apk, and the keyword features look for words associated with prizes, urgency, account verification and login. Keyword matching is only a heuristic and can give both false alarms and misses. A UPI QR code encodes a URI with parameters such as pa (payee address), pn (payee name) and mc (merchant code), and the three UPI features record whether these are present and well formed. The system does not check the payee or merchant against a bank or trusted registry, so it cannot confirm who owns the account.

VIII. MACHINE LEARNING MODEL

A Random Forest [7] is an ensemble of decision trees. Each tree is grown on a bootstrap sample of the training data and considers a random subset of features at every split. The forest pools the outputs of its trees, and the mean class probability across them serves as the machine-learning threat probability, P_{ML}. We chose it for three reasons: the inputs are tabular and of mixed scale, which tree models handle without normalisation; forests capture interactions such as an IP host combined with an executable extension; and they train quickly on small data. A feature-based model also keeps each decision tied to attributes that can be named in a warning.

The model was trained with scikit-learn [8] and saved and reloaded with Joblib. It uses 450 trees (n_estimators), a maximum depth of 12 and balanced class weights, which increase the influence of the smaller class. One seed (random_state = 42) was used for both the 80:20 stratified split and the classifier, so a run can be reproduced exactly. The input is the 23-feature vector and the output is the probability of the malicious class. The model is fast, needs no feature scaling and works offline. With only a few hundred samples, however, 450 trees of depth 12 can fit the training set almost exactly (training accuracy was 100%), and a forest cannot extrapolate to attack patterns it has never seen. That is why the final score also draws on rules. Only one learned model is used, so there is no ensemble of different algorithms. The score is computed as

$$\text{Risk Score} = \min(100, \max(0, \text{int}(0.7 \times P_{ML} \times 100 + 0.3 \times R_p))) \quad (2)$$

where P_{ML} is first converted to a whole-number percentage and R_p is the sum of the penalties from the rule-based checks. Table 2 lists them; the keyword penalty grows with the number of suspicious keywords found and is capped at 20. The result is clipped to the range 0–100 and truncated to a whole number, and a payload with a raw IP host or an executable extension is additionally raised to a score of at least 75. The 70/30 weighting is the one used in this implementation, and we did not test other weights. The classifier captures feature combinations learned from examples, whereas the rules encode security knowledge that is easy to explain and can lift the score for a payload the small training set represented poorly. Scores from 0 to 30 are labelled Safe, 31 to 70 Medium Risk and 71 to 100 High Risk.

Table 2. Rule-based penalties added to the risk score

Rule	Penalty
Raw IP address in the payload	+25
Link to an executable or script file	+35
URL shortener	+15
Suspicious top-level domain	+15
Suspicious keywords	+8 per keyword, at most +20

IX. SYSTEM ARCHITECTURE

The system has six layers, shown in Fig. 1. The input layer accepts an uploaded image, a camera capture or a test payload. The preprocessing and decoding layer recovers the payload with OpenCV and falls back to ZBar and ZXing-C++ when they are installed. The feature extraction layer types and parses the payload and builds the 23-feature vector. The machine learning and risk engine runs the Random Forest, applies the rule checks and computes the score through Equation (2). The persistence layer writes each scan to the SQLite database scan_history.db, and the presentation layer shows the payload, score, level and warnings in the Streamlit dashboard.

Fig. 1 shows these layers as implemented. The features in Table 1 are computed locally, with no external domain reputation service. Beside the score, the dashboard lists the indicators that were triggered, such as a raw IP address, an HTTP connection, a shortener, a suspicious TLD, an executable extension, several subdomains, suspicious keywords or incomplete UPI parameters, or states that all checks passed. These messages come from the rule-based checks and not from the internal reasoning of the Random Forest.

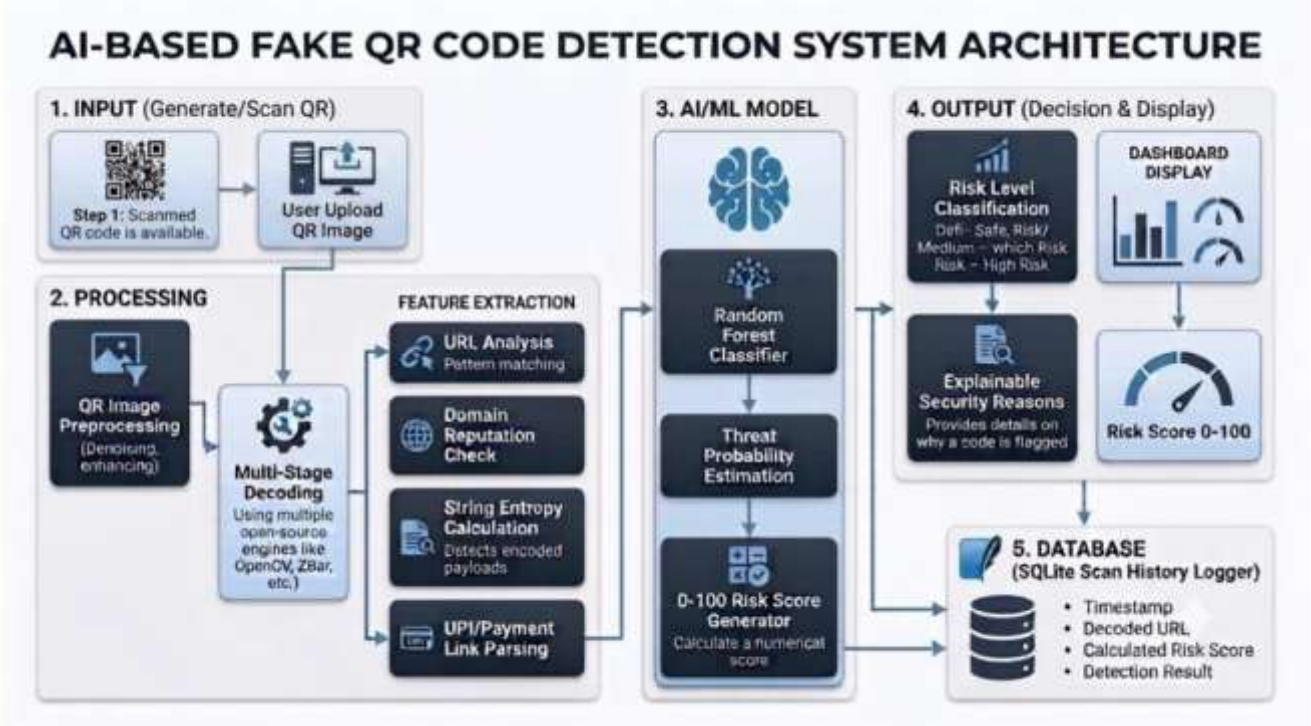


Fig. 1. Block diagram of the implemented system: input, decoding, feature extraction, ML and risk engine, SQLite persistence and Streamlit dashboard.

X. IMPLEMENTATION

The system is written in Python 3.12 or later and was developed in Visual Studio Code on Windows 11. OpenCV handles image loading, resizing, thresholding and the primary QR detector, with the optional pyzbar and zxing-cpp bindings as fallbacks. Scikit-learn trains and evaluates the Random Forest, Pandas and NumPy deal with the dataset and numeric work, Joblib stores the trained model, and the re, urllib.parse and math modules handle pattern matching, URL parsing and the entropy calculation. SQLite3 is the database engine and Streamlit provides the web interface.

The dashboard has five views: Scanner and Risk Engine, Threat Analytics, Audit Logs and Database, ML Model Architecture, and Academic Project Info. Users can upload an image, use the camera or pick a test vector. The audit log lists every scan with its timestamp, payload, type, score, level and reasons, and it can be filtered, exported as CSV and cleared. The dashboard accepts JPG, JPEG and PNG images up to 120 KB, while the project notes recommend 30–40 KB. The development machine had an Intel Core i3, i5 or i7 or AMD Ryzen 5 or 7 processor with 4 GB of RAM (8 GB recommended); no timing or memory measurements were taken. High-risk scans can be written to data/high_risk_samples.csv so that the next retraining includes them, but the model reported here was trained without any such samples. The dashboard is a demonstration prototype and not a hardened production service.

XI. EXPERIMENTAL RESULTS

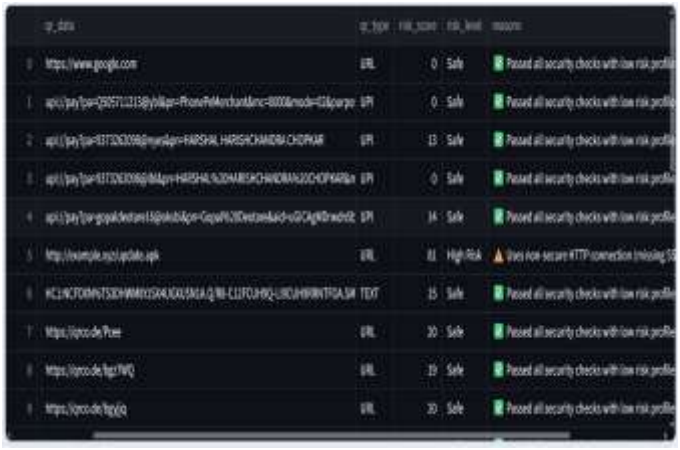
On the held-out 20% test partition (90 payloads: 59 safe and 31 malicious), the classifier reached 98.89% accuracy (Table 3). It made a single error, flagging one safe payload as malicious, and missed no malicious payload. For the malicious class, precision was 0.969, recall 1.00, F1-score 0.984 and ROC-AUC 1.00. Training accuracy was 100%, and five-fold stratified cross-validation on all 450 payloads gave 100% accuracy, precision, recall, F1-score and ROC-AUC in every fold. Because many payloads are near-copies and the classes differ in protocol (Section VI), these numbers show that the path from features to prediction works as designed, but not that the system would do as well on unseen QR codes. The most influential features in the forest were HTTPS (0.218), HTTP (0.168), Has_Suspicious_Keyword (0.110), Suspicious_Keyword_Count (0.109), Number_of_Subdomains (0.058) and Domain_Entropy (0.057). Two further checks show how easy the data are. With the HTTP and HTTPS features removed, five-fold accuracy stayed at 100%, so raw IP hosts, suspicious TLDs, keywords and executable extensions separate the classes just as well. After dropping the 75 repeated feature vectors (375 unique rows), five-fold accuracy was 99.73% (standard deviation 0.53) with all features and 100% without the protocol features.

Table 3. Classifier performance on the held-out 20% test partition

Metric	Result
Accuracy	98.89%
Precision (malicious class)	0.969
Recall (malicious class)	1.00
F1-score (malicious class)	0.984
ROC-AUC	1.00
Training accuracy	100%
Five-fold cross-validation accuracy	100%

Table 4 shows entries from the dashboard's audit log (scan history), captured from the running system. They are individual observed outputs and not a test set, so we draw no accuracy claim from them.

Table 4. Sample entries from the dashboard audit log (scan history)



In these entries, a well-known HTTPS site scored 0, UPI payment links scored between 0 and 14, a plain-text payload scored 15, and short HTTPS redirect links from a QR-generation service scored 19 or 20, all rated Safe. The one unencrypted link to an .apk file on a cheap TLD scored 81 and was rated High Risk. These outputs match the intended behaviour of the score.

To see how the model behaves on data it was not built around, we ran a supplementary check on a larger labelled collection of URLs (the file balanced_urls.csv). It contains 632,508 URLs, half benign and half malicious, with no UPI or plain-text payloads. We randomly drew 40,000 URLs per class (seed 42) and computed the same features. The model trained on our 450 payloads classified 86.33% of these 80,000 URLs correctly (Table 5, row A). It raised no false alarms on the benign half but missed 10,935 of the 40,000 malicious URLs, a recall of 72.66%. Retraining the same forest on 64,000 of these URLs and testing on the other 16,000 gave 98.94% accuracy (row B), but this result again leans on the protocol: in the collection 99.99% of the benign URLs begin with https://, whereas most malicious ones use http:// or have no scheme at all, and HTTPS and HTTP together account for 66% of the feature importance. Removing the two protocol features lowered accuracy to 87.69% (row C), which is closer to what the remaining features can achieve on their own.

Table 5. Supplementary check on URLs from the larger collection (precision, recall and F1-score for the malicious class)

Setting	Accuracy	Precision	Recall	F1-score	ROC-AUC
A. Model trained on 450 payloads, tested on 80,000 URLs	86.33%	1.000	0.727	0.842	0.971
B. Retrained on 64,000 URLs, tested on 16,000	98.94%	0.999	0.980	0.989	0.998
C. As B, without HTTP and HTTPS features	87.69%	0.975	0.774	0.863	0.947

XII. DISCUSSION

The pipeline behaved as intended, and in the logged examples the score separated the one clearly risky payload, an unencrypted link to an .apk file, from the ordinary ones, which all stayed at 20 or below.

The strong test results deserve careful reading, since accuracy is easy to inflate on a small set. Here the dataset is templated, a quarter of the test payloads have an exact twin in the training data, and the protocol alone almost separates the classes; even without the protocol features, keywords, raw IP hosts, TLDs and file extensions still separate them completely. A perfect 100% in five-fold cross-validation on the same data therefore says that the data are easy, not that the model is flawless. A forest of 450 trees with depth 12 can fit such data almost exactly, and the attack types in the data are the ones the authors thought of, so the test says little about obfuscated links, homograph domains or repeated redirects. The supplementary check points the same way: the model trained on the small set missed more than a quarter of the malicious URLs from the larger collection, and without the protocol features even a model trained on tens of thousands of URLs stayed below 88%. The results are best taken as a functional check and not as a claim about detection ability.

The qrco.de rows of Table 4 expose a practical weakness. Short redirect links from a QR-generation service scored 19 and 20 and were rated Safe, because the payload contains little that looks suspicious and the system does not follow redirects. A false positive, such as a genuine site on a cheap TLD, teaches users to ignore warnings, whereas a false negative, such as a phishing page on a normal-looking HTTPS domain or a fraudulent UPI address with well-formed parameters, is the more dangerous error. The UPI links in Table 4 were likewise rated Safe, but the system has no way to confirm who owns the payee account. The score should support the user's judgement and not replace it, and a deployed version would need a policy for access, retention and anonymisation of the scan log, since UPI payloads can contain personal details.

XIII. FUTURE SCOPE

None of the following exists in the current prototype. The first priority is a larger and more varied dataset with real QR samples, in which the safe and malicious classes cannot be separated by a few template cues such as protocol, keywords or raw IP hosts, evaluated with stratified cross-validation and a separate external test set. Domain age and WHOIS data, real-time reputation services, a verified merchant database and sandboxed redirect following would supply what payload text alone cannot. On the modelling side, a comparison with deep-learning models, feature-level explanations such as SHAP and periodic retraining on logged scans are natural next steps, and a mobile application would bring the check closer to the moment of scanning.

XIV. CONCLUSION

This paper presented a prototype that analyses QR payloads for security risk. It addresses the fact that a QR image reveals nothing about its destination and that ordinary scanners decode without judging. The prototype combines multi-pass decoding with OpenCV and optional ZBar and ZXing-C++ fallbacks, 23 payload-based features, a Random Forest classifier, rule-based penalties, a 0–100 QR Risk Score with three levels, readable warnings, SQLite logging and a Streamlit dashboard. The classifier reached 98.89% test accuracy and 100% in five-fold cross-validation on a dataset of 450 payloads, but the data are templated and easy to separate, and a supplementary check on 80,000 URLs gave only 86.3% for the same model. This shows that the pipeline works end to end, not how it would perform against real QR threats. The system is an aid that helps a user pause before opening a link or paying, not a guarantee against fraud. Future work should focus on a larger, more realistic dataset with proper validation and on external reputation and merchant-verification data.

ACKNOWLEDGMENT

The authors thank their guide, Dr. J. M. Bhattad, as well as the Head of Department, the Principal and the ETC laboratory staff of Priyadarshini College of Engineering, Nagpur, for their support.

REFERENCES

[1] A. Kharraz, E. Kirda, W. Robertson, D. Balzarotti, and A. Francillon, "Optical delusions: A study of malicious QR codes in the wild," in Proc. 44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN), 2014, pp. 192–203.

[2] M. A. Ozuberk and I. Cinar, "Malicious QR code classification using machine learning, deep learning, and explainable AI," in Proc. 2026 5th Int. Informatics and Software Engineering Conf. (IISEC), Ankara, Türkiye, 2026, pp. 128–133, doi: 10.1109/IISEC69317.2026.11418433.

[3] N. N. Ahamed, T. Murugan, and B. K. Mohanta, "QR code phishing (quishing): Food supply chain management traceability risks combined with blockchain technology," in Proc. 2025 3rd Int. Conf. Business Analytics for Technology and Security (ICBATS), 2025.

[4] A. S. Rafsanjani, N. B. Kamaruddin, H. M. Rusli, and M. Dabbagh, "QsecR: Secure QR code scanner according to a novel malicious URL detection framework," IEEE Access, vol. 11, pp. 92523–92539, 2023, doi: 10.1109/ACCESS.2023.3291811.

[5] J. Ma, L. K. Saul, S. Savage, and G. M. Voelker, "Beyond blacklists: Learning to detect malicious web sites from suspicious URLs," in Proc. 15th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining (KDD), Paris, France, 2009, pp. 1245–1254.

[6] C. E. Shannon, "A mathematical theory of communication," Bell Syst. Tech. J., vol. 27, no. 3, pp. 379–423, Jul. 1948.

[7] L. Breiman, "Random forests," Mach. Learn., vol. 45, no. 1, pp. 5–32, 2001.

[8] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," J. Mach. Learn. Res., vol. 12, pp. 2825–2830, 2011.