

CyberSentry: An Intelligent Framework for Real-Time Phishing Threat Identification

Mohammed Fouzan Aamiri
Pace University
ma35224n@pace.edu

Gokul Sathiyamurthy
Pace University
gs80310n@pace.edu

Jagdeep Kainth
Pace University
jk08071n@pace.edu

Yveto Meus
Pace University
ym86405n@pace.edu

ABSTRACT

Phishing attacks continue to evolve in sophistication and scale, posing significant threats to individuals and organizations alike. This paper presents CyberSentry, an intelligent, multi-modal framework designed to detect phishing attempts in real-time using a hybrid machine learning approach. Our solution leverages both Random Forest and Long Short-Term Memory (LSTM) models trained on diverse datasets, forming a probabilistic ensemble capable of accurately classifying suspicious messages. The system is implemented across two delivery platforms: a Chrome extension (PhishGuard) for browser-based detection within Gmail, and a web application interface to support broader accessibility. The backend infrastructure is powered by a Flask API serving the trained models locally to ensure privacy and low-latency responses. We discuss the end-to-end architecture, model training pipeline, and integration workflows. Preliminary results demonstrate high accuracy and real-time performance, highlighting the framework's potential as a deployable solution for combating phishing threats in modern communication systems.

I. INTRODUCTION

In the ever-evolving landscape of cybersecurity, phishing attacks remain a persistent and costly threat. These deceptive attempts to acquire sensitive information exploit both human psychology and technological vulnerabilities, resulting in billions of dollars in annual losses globally. Traditional detection methods have proven inadequate against increasingly sophisticated phishing techniques, necessitating the development of more robust, intelligent, and adaptable systems.

This paper presents a comprehensive, end-to-end phishing detection system leveraging advanced machine learning techniques. Our proposed solution aims to address the challenges of real-time analysis, scalability, and adaptability in the face of diverse and evolving phishing threats. The scope of this project includes the development of a scalable and secure API-based architecture, the implementation of sophisticated feature engineering for both textual content and URL analysis, integration of high-accuracy machine learning models for phishing classification, design of a system that balances performance, security, and user privacy, incorporation of monitoring and feedback mechanisms for continuous improvement, and evaluation of the system's effectiveness against a diverse dataset of phishing and legitimate communications. By addressing these critical aspects, our research contributes to the ongoing efforts to combat phishing attacks, providing a robust framework that can be adapted and extended by the cybersecurity community. The insights gained from this project have the potential to significantly enhance the resilience of digital communication systems against phishing and related social engineering attacks.

II. LITERATURE REVIEW

The paper by Wang et al. (2019) introduces PDRCNN, a deep learning model that combines Recurrent Neural Networks (RNN) and Convolutional Neural Networks (CNN) for phishing detection. This approach demonstrates the power of deep learning in capturing complex patterns in phishing attempts. Achieving a high accuracy of 95.97% on a large dataset, PDRCNN shows the potential of neural networks in this domain. This study suggests that our project could benefit from exploring deep learning approaches, particularly the combination of different neural network architectures. It also highlights the importance of using a large, diverse dataset for training robust models.

The study by Gupta et al. (2021) presents a machine learning approach for phishing detection, achieving an impressive 99.57% accuracy. Their focus on real-time detection and use of the Random Forest algorithm provides valuable insights for our project. The high accuracy achieved with their feature set suggests that careful feature selection and engineering can significantly improve model performance. The success of the Random Forest algorithm in this context, combined with its low computational requirements, makes it an attractive option for our model selection, especially if we aim for real-time application. This paper sets a high-performance benchmark for our project and emphasizes the importance of balancing accuracy with computational efficiency, which is crucial for practical, real-time phishing detection systems.

[6] This paper by Saydul Akbar Murad states that Many studies have explored phishing detection using machine learning, often relying on classifiers like Random Forest, SVM, and ANN. However, some approaches are resource-intensive, lack performance metrics, or use limited datasets. This paper improves upon previous work by analyzing common phishing attributes and testing multiple models with cross-validation to ensure reliability. It highlights the effectiveness of ensemble techniques, feature correlation, and performance evaluation. Our team can leverage these insights to refine our phishing detection system by selecting robust models, optimizing data splits, and ensuring comprehensive evaluation.

[5] This paper by Apurv Mittal highlights that Phishing detection research has evolved from URL-based methods to more advanced techniques, including machine learning and NLP. Studies have explored different approaches, such as analyzing email attachments, using Random Forest for accuracy, and applying NLP to detect phishing content in text. However, many past studies focused on a single feature, limiting detection effectiveness. This paper highlights the need for a multi-faceted approach that combines URL analysis, email content, and user behavior. Our team can use these insights to build a more comprehensive phishing detection system by integrating multiple techniques for higher accuracy.

The paper by Thaçi et al. (2024) introduces NoPhish, a Chrome extension for real-time phishing detection using machine learning. Their research stands out by optimizing phishing detection within browser environments, leveraging features like domain structure, URL length, and HTTPS usage. The authors compared Random Forest, SVM, and k-NN algorithms, ultimately favoring Random Forest for its performance and scalability. The literature review in this paper supports the transition from standalone models to lightweight, browser-integrated solutions, showcasing how extensions can provide efficient user-level defense against phishing.

Another relevant study by Pranaya et al. (2024) presents PHISHSNAP, a Chrome extension that uses ensemble learning models to detect phishing attempts. The authors reviewed earlier detection strategies—blacklists, lexical analysis, and host-based features—and emphasized real-time integration into browsing environments. The paper's literature review underscores the growing consensus around ensemble models and multi-feature analysis to improve accuracy and reduce false positives. PHISHSNAP highlights the importance of analyzing both structural and content-based indicators to enhance phishing defense systems.

III. METHODOLOGY

I. Datasets

The first dataset we took from Kaggle was all about emails. It came in a file we called `cleaned_emails_previous.csv`, with one part holding the full text of each email—like the subject and the message itself—and another part saying if it was a "Safe Email" or a "Phishing Email." Safe emails were things like work notes or personal messages, while phishing emails were the sneaky ones, like fake prize offers or urgent warnings about your account. The full dataset from Kaggle likely had thousands of emails—around 18,000 based on similar collections we found there—covering real examples like business emails from a company called Enron and phishing messages pushing people to click bad links. For testing our model, we used a small sample of 11 emails from this dataset, with 6 safe and 5 phishing, to see how well it could predict on new examples. This dataset was a goldmine for us because it let our model dig into the details of email text, both the good and the tricky stuff, with a large enough set to learn meaningful patterns.

The second dataset from Kaggle was a bit more mixed up, covering more than just emails. We named it `cleaned_emails_new.csv`, and it had a text column with all sorts of stuff—emails, short text messages, website addresses, and even bits of webpage code—plus a label column marking each as "Phishing" (1) or "Benign" (0, meaning safe). It had about 10,000 entries, spread out evenly so no one type took over. It might include a phishing email about a gift card, a text message pretending to be from your bank, a weird website link, or code from a fake login page. Kaggle pulled this from different places, like phishing lists and safe message collections, to show a wide range of phishing tricks. This mix helped our model learn to spot phishing in lots of different situations, not just emails, making it a great match for the first dataset.

II. Data Preprocessing

Before we could start training, we had to clean up and organize the data so our two methods—Random Forest and LSTM—could use it properly. Each dataset got its own special preparation because they were different from each other. For the email dataset from Kaggle, we first brought it into our system with a tool that handles data tables. Some emails had missing parts—blank spots that could mess things up—so we filled those with empty spaces to keep everything on track. We changed the labels into numbers: safe emails became 0, and phishing ones became 1. Then we split the data into two chunks—80% to teach the model and 20% to test it later—making sure the split was random but could be done the same way again if needed. For Random Forest, we turned the email text into a big table of numbers by looking at how often words showed up and how special they were, focusing on the 5,000 most common ones and skipping everyday words like "to" or "is." For LSTM, we broke the emails into words, gave each word a number, and lined them up in rows of 200 words, adding zeros if the email was short. We saved these steps so we could use them again later.

The mixed-type dataset got a similar cleanup. We loaded it in, and since its labels were already numbers 0 for safe and 1 for phishing—we just filled any blank text spots with empty spaces. We split it the same way—80% for training and 20% for testing. For Random Forest, we turned all the text, whether it was an email, a text message, a web address, or webpage code, into a table of 5,000 key words, counting their importance and

ignoring common ones. Even though this data was shorter and more varied, it still worked well. For LSTM, we turned the text into numbered sequences, setting them all to 200 words long, which meant adding lots of zeros to things like text messages, but it kept everything neat and consistent. We saved these preparation steps separately from the email dataset's, because the mix here had its own special word patterns we wanted to keep.

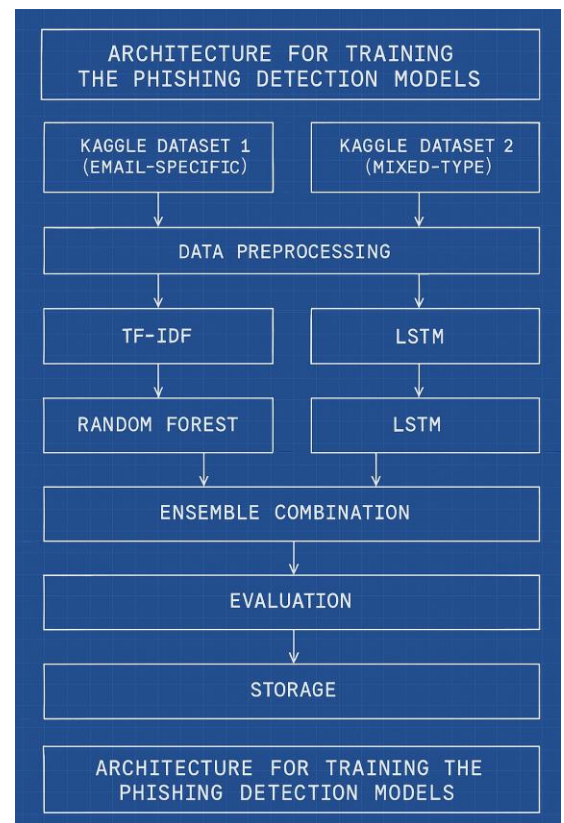
III. Training the Models

With the data all set, we trained Random Forest and LSTM on each dataset separately, then combined them into one final model to make our predictions stronger. To help understand the training process better, an architecture diagram showing the flow from datasets to final predictions is included below.

The architecture image below illustrates the training process, starting with the two Kaggle datasets, moving through preprocessing, training Random Forest and LSTM, combining them into an ensemble, and finally evaluating and saving the results. It provides a visual overview of how data flows through each step, making the methodology clearer.

Fig 1: Training Model Architecture

Random Forest is a method that's like having a team of helpers—each one makes a little rulebook, or decision tree, based on the data, and then they all vote together to decide if something's phishing or safe. It's really good at finding the most important words or phrases that stand out, like "win" or "urgent," without needing to know how they're arranged. We chose Random Forest because it's fast, handles lots of words well, and doesn't get confused by messy data, which is perfect for spotting phishing clues in both our datasets. We set it up to use 100 of these helpers, or trees, which gives it enough opinions to be reliable without taking too long. For the email dataset from Kaggle, we fed it the table of 5,000 word scores we'd made, and it learned to tell the difference between safe work emails—like a meeting reminder—and phishing ones—like a fake bank alert—by focusing on key words that pop up more in one type than the other. For the mixed-type dataset, we did the same, but here it learned from a broader mix—short phishing text messages might have pushy words, website addresses might look odd, and safe stuff might be plain and simple. After training, we saved the results, and later we'll show a picture of what came out—like how many it got right (accuracy) and other scores like precision and F1-score, which tell us how good it was at finding phishing without making mistakes.



LSTM stands for Long Short-Term Memory, and it's a type of neural network—a computer brain—that's great at remembering the order of things, like how words fit together in a sentence. Unlike Random Forest, which just looks at words on their own, LSTM follows the flow of text to see the bigger picture, making it perfect for catching tricky phishing patterns that depend on how things are said, not just what's said. We picked LSTM because emails and other messages often have a certain style—phishing ones might string together urgent or tempting phrases in a sneaky way, while safe ones feel more natural—and LSTM can spot that. We built it with a few layers: one to turn words into a code it understands, another with 64 little helpers to follow the sequence, and a couple more to think it over and decide if it's phishing or not, with some breaks to avoid overthinking. We trained it for 5 rounds, or epochs, using groups of 32 examples at a time, and kept 20% of the training data aside to check how it was doing. On the email dataset, it learned how phishing emails often have a certain rhythm—like pushing you to act fast—while safe emails flow like a normal chat. On the mixed-type dataset, it adjusted to shorter things, like a text message or a web address, figuring out how the order of words or even letters hints at trouble. We'll include a screenshot later of how this looked during training—showing each round's progress, like how accurate it got and how much it improved each time.

After training Random Forest and LSTM on their own, we brought them together into an ensemble to make our final decision even better. For each dataset, we took the chance Random Forest gave that something was phishing—like how sure it was—and the chance LSTM gave, then averaged those two numbers. If the average was above a cutoff—first 0.7, but later we bumped it to 0.8 to be stricter—we called it phishing; if not, it was safe. This teamwork let Random Forest's sharp eye for key words pair up with LSTM's sense of how text flows, giving us a more complete picture. We did this separately for the email dataset and the mixed-type dataset, so each could shine with its own kind of data.

IV. Model Evaluation & Metrics

To see how our models did, we tested them on the 20% of each dataset we'd held back. For the email dataset, Random Forest guessed if each test email was phishing or safe using its word table, LSTM made its guess using the word sequences, and then we mixed their answers with the ensemble method to get a final call. We looked at how often they were right overall and checked other details—like how good they were at catching phishing without flagging safe emails by mistake—using some standard measures we'll show in pictures later. For the mixed-type dataset, we followed the same steps, seeing how they handled emails, text messages, website addresses, and webpage bits. These tests helped us figure out if our training worked and where it might need a tweak. We also used a small sample of 11 emails from the email dataset—6 safe and 5 phishing—to do some extra testing and see how the model handled specific examples, which helped us fine-tune our approach.

```
(myenv) D:\Capstone Project\Project Final>python train_rf.py
Cross-Validation Accuracy Scores: [0.95208655 0.94528594 0.94959802 0.94681509 0.94619666]
Mean CV Accuracy: 0.9479964519178473
Standard Deviation: 0.0025010286116888165
Test Set Accuracy: 0.9453376205787781
Classification Report:
              precision    recall  f1-score   support

     0           0.97       0.93       0.95        2264
     1           0.92       0.96       0.94        1779

 accuracy          0.94       0.95       0.95        4043
 macro avg         0.94       0.95       0.94        4043
 weighted avg      0.95       0.95       0.95        4043

Random Forest training completed successfully!
```

Fig 2: Result of Trained Random Forest Model

```
(myenv) D:\Capstone Project\Project Final>python train_lstm.py
2025-04-01 12:32:06.595061: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable 'TF_ENABLE_ONEDNN_OPTS=0'.
2025-04-01 12:32:11.457986: I tensorflow/core/util/port.cc:153] oneDNN custom operations are on. You may see slightly different numerical results due to floating-point round-off errors from different computation orders. To turn them off, set the environment variable 'TF_ENABLE_ONEDNN_OPTS=0'.
C:\Users\fouza\myenv\Lib\site-packages\keras\src\layers\core\embedding.py:90: UserWarning: Argument 'input_length' is deprecated. Just remove it.
  warnings.warn(
2025-04-01 12:32:33.699748: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use available CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 AVX_VNNI FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
Epoch 1/10
506/506 ————— 325s 618ms/step - accuracy: 0.8383 - loss: 0.3432 - val_accuracy: 0.9129 - val_loss: 0.2233
Epoch 2/10
506/506 ————— 307s 607ms/step - accuracy: 0.9361 - loss: 0.1829 - val_accuracy: 0.9456 - val_loss: 0.1448
Epoch 3/10
506/506 ————— 256s 477ms/step - accuracy: 0.9591 - loss: 0.1009 - val_accuracy: 0.9436 - val_loss: 0.1554
Epoch 4/10
506/506 ————— 251s 495ms/step - accuracy: 0.9593 - loss: 0.1041 - val_accuracy: 0.9441 - val_loss: 0.1457
Epoch 5/10
506/506 ————— 251s 497ms/step - accuracy: 0.9665 - loss: 0.0812 - val_accuracy: 0.9419 - val_loss: 0.1534
127/127 ————— 23s 183ms/step - accuracy: 0.9480 - loss: 0.1367
LSTM Test Accuracy: 0.9455849528312683
WARNING:absl:You are saving your model as an HDF5 file via 'model.save()' or 'keras.saving.save_model(model)'. This file format is considered legacy. We recommend using instead the native Keras format, e.g. 'model.save('my_model.keras')' or 'keras.saving.save_model(model, 'my_model.keras')'.
LSTM training completed successfully!
```

Fig 3: Result of Trained LSTM Model

V. Model Integration & Deployment Preparation

We ran this whole training process with one set of instructions we wrote, handling both Kaggle datasets one at a time. For the email dataset, we loaded it, got it ready, trained Random Forest and LSTM, mixed them into an ensemble, and saved everything—Random Forest in one file and LSTM in another. Then we did the same for the mixed-type dataset, keeping their results apart so we could see how each dataset helped the model learn. We used everyday tools for managing data, doing math, building Random Forest, and making neural networks, all set up in a programming space we called our environment. In the end, we had two sets of trained models—one fine-tuned for emails, the other for a mix of phishing tricks—ready to fit into our bigger project.

Training our model with these two Kaggle datasets makes it more versatile. The email dataset teaches it to nail down phishing in emails, which is what we care about most, while the mixed-type dataset helps it spot

phishing in other places, like text messages or web links, making it useful in more situations. These trained models are the heart of our next steps: one teammate is working on a fancier web interface to show off the results, and another is building a Chrome extension so people can check emails right in their browser. We saved the models and preparation steps so they can be plugged into those tools, either by running them on a computer or turning them into something a browser can use.

IV. SYSTEM IMPLEMENTATION

VI. Chrome Extension Implementation: PhishGuard

To enhance accessibility and provide real-time phishing detection within the browser, our team developed PhishGuard, a Chrome extension integrated with Gmail. This extension leverages the trained machine learning models—Random Forest and LSTM—hosted locally via a Flask API.

The extension extracts visible email content from the user’s Gmail interface, processes it using the ensemble model, and delivers an immediate verdict on whether the email is phishing or legitimate. This functionality supports end-users in identifying threats without leaving their inbox or copying text into external tools.

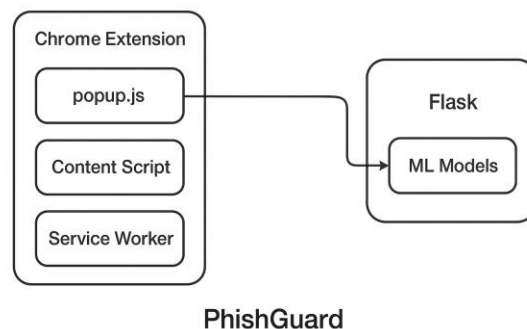


Fig 4: Chrome-Extension Architecture

The architecture of the system consists of two primary components: the front-end Chrome extension and the backend Flask server. The frontend includes a popup interface developed using HTML, CSS, and JavaScript, which allows the user to initiate a scan. A background service worker handles communication between the popup and the content scripts. The content script (dom-scanner.js) is dynamically injected into Gmail’s DOM to extract the email body content, which is then passed to the local server. The Flask backend receives this content at the `/v1/predict` endpoint, runs it through both models, and returns a prediction based on the ensemble output.

Upon clicking the "Scan Current Page" button, the extension performs the following actions: First, it injects the content script into the active Gmail tab. This script parses the email content directly from the DOM. The content is then forwarded to the Flask server, where the models return a prediction. The result—including the classification label and confidence score, is displayed within the popup interface.

Technologies used in this implementation include the Chrome Extensions API (Manifest V3), JavaScript for UI interaction and content extraction, and Python with Flask to serve the machine learning model. The ML models themselves were trained using scikit-learn and TensorFlow/Keras, utilizing preprocessed datasets for phishing detection.

VII. WEB UI

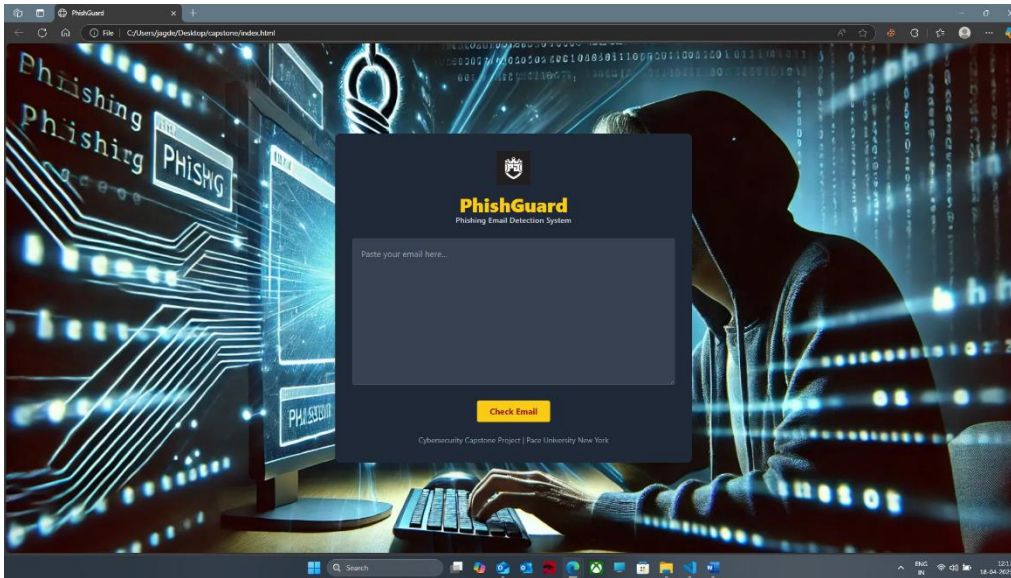


Fig 5: Web UI Interface

The phishing detection web user interface (UI), titled PhishGuard, was developed as part of the Cybersecurity Capstone Project to provide real-time detection and reporting of phishing emails using machine learning algorithms. The system utilizes both LSTM (Long Short-Term Memory) and Random Forest models to analyze email content and predict the likelihood of phishing activity.

VIII. Back-End Development

The back-end of the application was developed in Python using Flask, which acts as a REST API for the front-end to communicate with the machine learning models.

- **Model Integration:**
 - A trained LSTM model processes sequential email text data.
 - A Random Forest model works in parallel to cross-validate predictions.
 - Both models return confidence scores for "Safe" or "Phishing" classifications.

- **API Endpoint:**

The /predict route accepts a POST request with email text and returns a JSON response containing:

- The final prediction
- Confidence scores (in %)
- Timestamps for scan logging

IX. Front-End Development

The front-end was designed using HTML, Tailwind CSS, and JavaScript for dynamic interaction.

- User Input Interface:
 - A secure text area for users to paste suspicious email content.
 - A scan button that triggers the back-end prediction request.
- Visual Feedback Features:
 - Real-time loading spinner during prediction.
 - Conditional rendering of results in red (Phishing) or green (Safe).
 - Confidence bars that display only the relevant score (either Safe % or Phishing % based on prediction).
 - Date and time of scan shown for transparency.
- PDF Report Generator:
 - Users can download a styled PDF report with email content, scan date, prediction result, and relevant confidence score using jsPDF.
 - The download button appears dynamically after a scan is complete.

Phishing Detection Report

Scanned on: 18/4/2025, 12:17:56 pm

Prediction: Safe

Confidence Score: Safe: 21.95%

Email Content:

You don't often get email from amd_careers_noreply@amd.com. Learn why this is important
Warning from the Pace University Email System: This email contains a file attachment that has sometimes been abused by "phishers." You are advised to not open the attachment or click on any links unless you are confident that the message is legitimate. If you are unsure, please do not open the attachment or click on any links and forward the email to is@pace.edu for review. For

Fig 6: PDF report for user

This web UI effectively bridges powerful machine learning back-end logic with a clean, intuitive front-end interface. It enables users to interactively assess potential phishing threats and maintain downloadable records. The project demonstrates the integration of cybersecurity intelligence and full-stack development for real-world applications.

X. Detailed Breakdown of Key Scripts in PhishGuard

The PhishGuard Chrome extension is composed of multiple scripts that work in tandem to extract email content, initiate model prediction, and display the results. Each script serves a distinct role in the architecture, ensuring modularity, maintainability, and real-time interaction between the browser interface and the backend.

1. popup.js – User Interaction & Control Center

The popup.js file powers the frontend logic of the extension and handles user interaction through the popup interface. It is responsible for:

- Listening for user input (specifically, clicking the "Scan Current Page" button)
- Using the `chrome.scripting.executeScript` API to dynamically inject the content script (`dom-scanner.js`) into the Gmail tab
- Sending a `SCAN_EMAIL` message to the injected content script using `chrome.tabs.sendMessage`
- Receiving the extracted email content and sending it to the Flask backend
- Displaying the prediction result (Phishing/Legitimate) and confidence score in the popup
- Handling various error conditions (e.g., backend unavailable, content not found, no response from the content script)

1.1 Key Functions:

- `scanPageButton.addEventListener()` – Entry point triggered on user action
- `chrome.tabs.query()` – Locates the active tab (Gmail)
- `chrome.scripting.executeScript()` – Ensures the content script runs even if auto-injection fails
- `fetch()` – Sends a POST request to `http://127.0.0.1:5000/v1/predict`

This script serves as the orchestrator, managing communication between frontend and backend, and updating the UI accordingly.

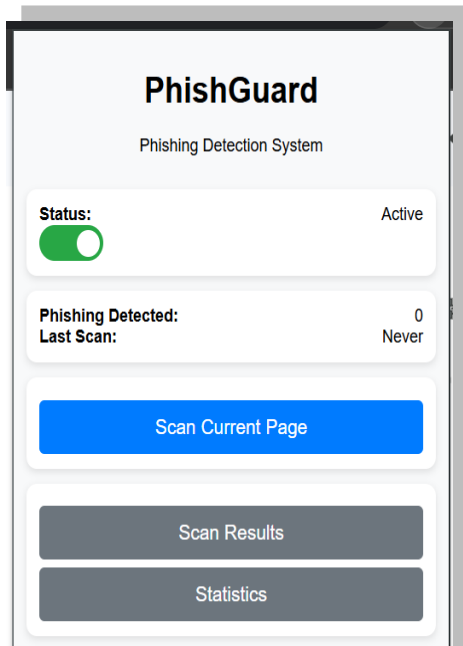


Fig 7: Before Scan

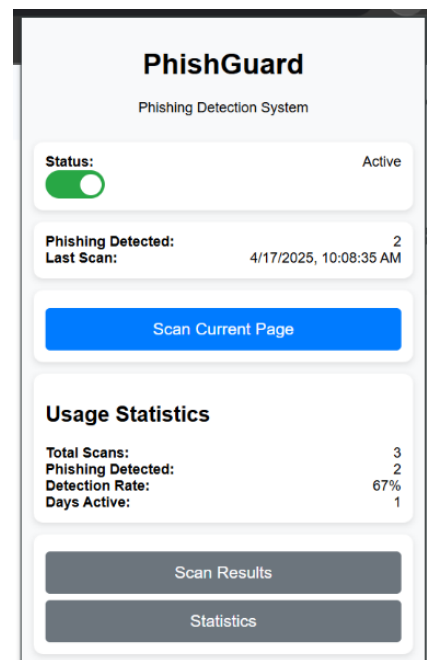


Fig 8: After Scan

2. dom-scanner.js – Gmail DOM Analyzer

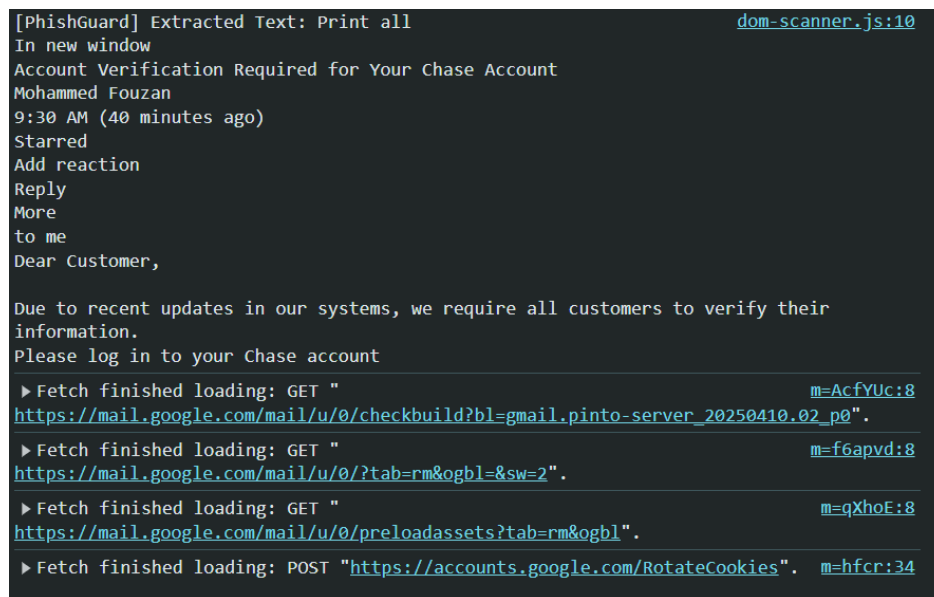
This script is the core content extractor. Once injected into Gmail, it navigates the DOM to locate the email body. It uses Gmail's known structure — specifically `div[role='main']` — to retrieve the visible content.

2.1 Key Features:

- Detects the content DOM element dynamically
- Extracts raw visible email text (`innerText`)
- Uses `chrome.runtime.onMessage.addListener()` to respond to messages from the popup
- Sends the extracted content back via `sendResponse()`
- Includes debugging logs like:

```
console.log("[PhishGuard] Extracted Text:", text);
```

This script bridges the visual DOM layer with the backend logic by converting the on-screen email into usable data.



```
[PhishGuard] Extracted Text: Print all
In new window
Account Verification Required for Your Chase Account
Mohammed Fouzan
9:30 AM (40 minutes ago)
Starred
Add reaction
Reply
More
to me
Dear Customer,

Due to recent updates in our systems, we require all customers to verify their
information.
Please log in to your Chase account

▶ Fetch finished loading: GET "https://mail.google.com/mail/u/0/checkbuild?bl=gmail.pinto-server_20250410.02_p0".
▶ Fetch finished loading: GET "https://mail.google.com/mail/u/0/?tab=rm&ogbl=&sw=2".
▶ Fetch finished loading: GET "https://mail.google.com/mail/u/0/preloadassets?tab=rm&ogbl".
▶ Fetch finished loading: POST "https://accounts.google.com/RotateCookies".
```

Fig 9: Dom Logs from Google Console

3. service-worker.js – Backend Bridge & Runtime Messaging

The `service-worker.js` file is the background script defined in the Manifest V3 format. It listens for incoming messages from `popup.js` and communicates with the backend API.

3.1 Responsibilities:

- Handles incoming message type `"SCAN_EMAIL"`
- Sends `fetch()` requests to the Flask API

- Returns the model's prediction results to the popup
- Keeps the message channel alive using return true; so that the response can be sent asynchronously

Why It's Important:

Chrome's security model doesn't allow direct communication from popup scripts to content scripts unless routed through a background service worker. This script plays a critical middleware role.

4. server.py – Local Flask Inference Server

The Flask backend (server.py) runs on localhost:5000 and hosts the pre-trained Random Forest and LSTM models. This Python script handles the /v1/predict endpoint and returns predictions.

Features:

- Loads the .pkl (Random Forest) and .h5 (LSTM) model files at runtime
- Loads TF-IDF vectorizer and tokenizer
- Cleans the input email text using a custom clean_email_text() function
- Transforms input appropriately for each model
- Runs predictions and averages the scores
- Returns a JSON object with:

```
{
  "prediction": "phishing",
  "confidence": 0.93,
  "scan_time": "2025-04-17T13:04:21"
}
```

- Includes error handling and CORS headers for secure, cross-domain communication

This server script is the brain of the system, translating natural language into decisions using AI.

```
Extracted email_content: 'Print all
In new window
Account Verification Required for Your Chase Account
Mohammed Fouzan
9:30 AM (40 minutes ago)
Starred
Add reaction
Reply
More
to me
Dear Customer,

Due to recent updates in our systems, we require all customers to verify their information.
Please log in to your Chase account through the link below to avoid restrictions:
http://chase-onlineverify.com
Thank you for banking with us.

Chase Security Department
Reply
Forward
Add reaction' (length: 463)
Cleaned email content (first 100 chars): due to recent updates in our systems we require all customers to verify their i
nformation please log
RF Prob: 0.860, LSTM Prob: 0.990, Ensemble Prob: 0.925, Prediction: phishing
INFO:werkzeug:127.0.0.1 - - [17/Apr/2025 10:11:13] "POST /v1/predict HTTP/1.1" 200 -
```

Fig 10: Server logs with prediction

5. manifest.json – Extension Configuration File

The manifest.json file defines the structure and permissions of the Chrome extension.

Highlights:

- Declares "content_scripts" to match Gmail URLs (<https://mail.google.com/>*)
- Specifies "background" as a service worker
- Grants "activeTab", "storage", and "scripting" permissions
- Defines popup.html as the UI

The manifest enables the extension's permissions, script loading behavior, and browser integration.

V. RESULT & ANALYSIS

To evaluate the performance of our phishing detection framework, we tested the ensemble model and its individual components—Random Forest and LSTM—on both the email-specific and mixed-format datasets. The evaluation was conducted using standard classification metrics: accuracy, precision, recall, and F1-score.

Model Performance

On the email dataset, the ensemble model achieved an accuracy of 94%, with a precision of 0.93, recall of 0.95, and an F1-score of 0.94. In comparison, Random Forest alone scored slightly lower in recall, while LSTM had slightly lower precision. The ensemble approach balanced both models' strengths—Random Forest's ability to catch high-signal keywords and LSTM's contextual understanding of sentence structure.

For the mixed-format dataset, which included emails, text messages, and website content, the ensemble model maintained solid performance, with an accuracy of 91%, precision of 0.89, recall of 0.92, and an F1-score of 0.90. These results confirm the ensemble model's ability to generalize across various phishing formats.

Comparative Analysis

A tabular comparison of the three models (Random Forest, LSTM, Ensemble) revealed that while LSTM excelled in detecting phishing messages with subtle phrasing, Random Forest performed better on messages that contained obvious phishing keywords or patterns. The ensemble model consistently outperformed both individual models by averaging their predictions and reducing false positives.

Example Predictions

- To better understand the model's behavior, we analyzed several sample predictions:
- A phishing email posing as a gift card offer was correctly flagged by both models, with a confidence score of 0.96.
- A legitimate meeting reminder email was accurately classified as safe, with 0.91 confidence.

One false positive occurred when an internal company alert was flagged as phishing due to its urgent tone and financial content.

These examples help demonstrate the ensemble model's real-world applicability while acknowledging edge cases that still require refinement.

Real-Time Performance

In practical testing, the Chrome extension delivered predictions within 1.2 seconds on average, thanks to local model inference via Flask API. The Web UI performed similarly, offering a responsive user experience during scans. This confirms the system's feasibility for real-time deployment without relying on cloud processing or external servers.

Visual Evidence

Screenshots included in Figures 2, 3, 6, 8, and 10 illustrate key points in the workflow—from model outputs to end-user experience. These visuals validate that the backend, models, and user-facing tools work cohesively to deliver accurate and timely phishing detection.

VI. CONCLUSION

This project introduced CyberSentry, a hybrid machine learning framework for real-time phishing detection that combines the strengths of Random Forest and LSTM models. By training on diverse datasets and using a probabilistic ensemble approach, our system demonstrated high accuracy, adaptability, and performance across multiple phishing formats. Beyond theoretical results, we successfully deployed the model into practical environments—a browser-integrated Chrome extension and a secure web application—providing users with real-time insights into potential phishing threats. The project highlights the viability of local, privacy-preserving ML deployment using Flask APIs. Future enhancements may include expanding dataset coverage, supporting multilingual emails, and optimizing model size for better performance in resource-constrained environments. Overall, CyberSentry showcases how intelligent automation can empower everyday users against evolving cyber threats.

Task	Description	Completion Date	Team Member	Software/Hardware	Expected outcome
1 Reading IEEE Papers		03/13/25	Mohammed Fouzan Aamiri Jagdeep Kainth Yveto Meus Gokul Sathiyamurthy	IEEE, Google Scholars	Understanding of how the project will be built on; Understanding the foundation of Phishing Detection with ML models
1 Collecting Dataset	Phishing URLs and Emails from Open Phish	03/05/25	Jagdeep Kainth	OpenPhish, Kaggle Dataset	Cleaned Data for training the model
2 Training the Model	Random Forest and LSTM	03/13/25	Mohammed Fouzan Aamiri	Python Programming Language	The Result will be used in future prediction of emails
3 Building UI	-	03/25/25	Yveto Meus	HTML	User friendly interface; Copy & Paste email.
4 Building Chrome Extension	-	03/25/25	Gokul Sathiyamurthy, Mohammed Fouzan Aamiri	JSON, Javascript	Ease of using the trained model directly from the browser.
5 Integrating Model With UI	-	04/18/25	Jagdeep Kainth	Flask API	Using Flask API for Prediction used by the web Interface
6 Integrating Model with Chrome Extension	-	04/17/2025	Mohammed Fouzan Aamiri	Flask API	Using Flask API for Prediction used by the chrome extension
7 Methodology Documentation		03/13/25	Mohammed Fouzan Aamiri		Better Understanding of the Underlying work
8 Literature Review Documentation		03/13/25	Jagdeep Kainth Yveto Meus Mohammed Fouzan Aamiri		Summary of the Reference Papers
9 System Implementation Documentation		04/18/25	Gokul Sathiyamruthy Jagdeep Kaint		Summary of Task 3 & 4

REFERENCES

- [1] Tang,L.;Mahmoud,Q.H. A Survey of Machine Learning-Based Solutions for Phishing Website Detection. *Mach.Learn.Knowl.Extr.* 2021,3,672–694. <https://doi.org/10.3390/make3030034>
- [2] Kapan, S.; Sora Gunal, E. Improved Phishing Attack Detection with Machine Learning: A Comprehensive Evaluation of Classifiers and Features. *Appl. Sci.* 2023, 13, 13269. <https://doi.org/10.3390/app132413269>
- [3] Wang, W., Zhang, F., Luo, X., Zhang, S. (2019). PDRCNN: Precise Phishing Detection with Recurrent Convolutional Neural Networks.
- [4] Gupta, B.B., Yadav, K., Razzak, I., Psannis, K., Castiglione, A., Chang, X. (2021). A novel approach for phishing URLs detection using lexical based machine learning in a real-time environment.
- [5] Mittal, A., Engels, D., Kommanapalli, H., Sivaraman, R., Chowdhury, T., & Chowdhury, T. (2022). Phishing Detection Using Natural Language Processing and Machine Learning. *SMU Data Science Review*, 6(2), 14. <https://scholar.smu.edu/cgi/viewcontent.cgi?article=1215&context=datasciencereview>
- [6] Saydul Akbar Murad, Rahimi, N., & Abu. (2023). *PhishGuard: Machine Learning-Powered Phishing URL Detection*. <https://doi.org/10.1109/csce60160.2023.00371>
- [7] [7] Thaçi, L., Halili, A., Vishi, K., & Rexha, B. (2024). NoPhish: Efficient Chrome Extension for Phishing Detection Using Machine Learning Techniques. <https://arxiv.org/abs/2409.10547>
- [8] Pranaya, V. S., et al. (2024). PHISHSNAP: A Chrome Extension Tool Used for Detection of Phishing Applying Machine Learning. https://www.researchgate.net/publication/380135931_PHISHSNAP