PhishBait

AI-Based In-Game Chat Phishing Detection

AI&NN (20CYS304) Mini Project

ANAGHA B PRASANTH - CB.SC.U4CYS23002
DEVINANDHA - CB.SC.U4CYS23011
ISHITHA PRAVEEN - CB.SC.U4CYS23018

PROBLEM STATEMENT

Online multiplayer games have witnessed rapid growth, leading to large and diverse player communities that engage through open chat systems. However, these chat environments are increasingly exploited by malicious attackers who send phishing messages, scam links, or fake promotions to trick players into revealing personal information or visiting harmful websites. Players are often easily deceived by such fake offers or links, especially when messages appear casual or game-related. Traditional keyword-based filters and detection methods frequently fail in this context, as in-game chats often use highly informal, abbreviated, or context-specific language.

Therefore, there is a critical need for a real-time detection system that can locally monitor in-game chat, identify potential phishing or scam messages using AI-based models, and instantly alert players without interrupting gameplay or affecting system performance.

PROPOSED SOLUTION

The proposed solution is to design and develop an AI-based real-time phishing detection system for online multiplayer game chat environments. The system aims to automatically identify phishing or scam messages and notify users without disrupting their gaming experience. This will be achieved through the integration of a browser extension that captures in-game chat messages and forwards them to a locally hosted Flask service. The Flask service will process the messages using a trained Multilayer Perceptron (MLP) model to classify them as legitimate or phishing attempts and return the results to the extension for user notification. The ultimate goal is to ensure a safer and more secure gaming environment by enabling real-time detection of phishing messages.

IMPLEMENTATION DETAILS

ALGORITHM

Step 1: Initialize Flask application

- Import required modules (Flask, request, jsonify, CORS, threading, sklearn exceptions, your phish_detector module).
- Create app = Flask(name) and enable CORS.

Step 2: Start background model training on server boot

- Define start_training_thread() that spawns a daemon thread calling train_model().
- Call start_training_thread() in if name == "main": before app.run().

Step 3: Expose /detect POST endpoint

- Define @app.route("/detect", methods=["POST"]) that: parse JSON payload, validate text field, call predict(text), handle NotFittedError (503) and generic exceptions (500), and return JSON result.

Step 4: Expose /status GET endpoint

- Define @app.route("/status", methods=["GET"]) to return {"ready": _model is not None} so clients can check model readiness.

Step 5: Configure constants and dataset paths

- Set DATASETS_DIR, CSV_FILES, SUSPICIOUS_KEYWORDS, URL_PATTERN, and KEYWORD_TOP_K as module-level constants.

Step 6: Define global model objects and lock

- Initialize _model = _vectorizer = _label_encoder = None and _model_lock = threading.Lock() to ensure thread-safe predictions.

Step 7: Implement dataset loader load_datasets()

- For each CSV in CSV_FILES: build full path, skip missing files, read chat_message and category, rename to text/label, map labels (spam→phishing, not_spam→safe), concatenate DataFrames, raise FileNotFoundError if none found, return combined DataFrame.

Step 8: Build heuristic feature extractor extract_features(text)

- Compute: keyword_count (suspicious keywords present), has_url (URL regex match), exclamations (count !), questions (count ?), all_caps_ratio (ratio of all-uppercase words to total words), and length (characters). Return features dict.

Step 9: Create TF-IDF + MLP training routine train_model()

- Load datasets, encode labels with LabelEncoder, build TfidfVectorizer(ngram_range=(1,2), max_features=15000, stop_words="english"), create MLPClassifier(hidden_layer_sizes=(128,64), max_iter=30, random_state=42, verbose=True), compose pipeline with make_pipeline(), fit pipeline on texts and labels, set _model, _vectorizer, _label_encoder, and print summary.

Step 10: Provide extract_keywords(text, top_k) utility

- If _vectorizer is available, transform text to TF-IDF vector, get feature_names, sort feature weights descending, return top non-zero terms up to top_k.

Step 11: Implement predict(text) with model + heuristics

- Acquire _model_lock, raise NotFittedError if _model is None, compute _model.predict_proba([text]) to get phishing_prob, compute feats = extract_features(text).

Step 12: Compute heuristic boost and final score

- Define boost = 0.2*(keyword_count>0) + 0.3*has_url + 0.1*(all_caps_ratio>0.5) + 0.1*((exclamations>2) or (questions>2)).
- Compute final score = min(phishing_prob + boost, 1.0). Set label = "phishing" if score >= 0.5 else "safe".

Step 13: Return rich prediction payload

- Return JSON-like dict with keys: label, score (rounded), keywords (from extract_keywords), raw_prediction (phishing_prob formatted), and heuristics (feats).

Step 14: Write Flask server main entry

- In if name == "main": call start_training_thread(), print server info, and app.run(host=HOST, port=PORT, debug=False).

Step 15: Define extension manifest metadata (manifest.json)

- Set manifest_version: 3, name: "PhishBait", version, description, and permissions: ["storage","activeTab","scripting"].
- Add content_scripts with matches for ://skribbl.io/ and http://localhost:3000/* and js: ["content.js"].

Step 16: Define content script constants and server URL (content.js)

- Set const SERVER_URL = "http://127.0.0.1:5000/detect" and define container selectors used by game handlers.

Step 17: Implement sendToServer(text) helper in content script

- Async POST JSON { text } to SERVER_URL, return parsed JSON on success, return null on network or non-2xx errors.

Step 18: Implement UI badge helper showBadge(targetSpan, result)

- Return early if no targetSpan or result or if targetSpan.dataset.phishChecked is set.
- Mark as checked, create with label and rounded percent, style based on result.label (phishing vs safe), append to targetSpan, and schedule removal after 12 seconds.

Step 19: Implement message de-duplication sets for games

- Create hangmanSeen = new Set() and skribblSeen = new Set() to avoid reprocessing identical messages.

Step 20: Implement Hangman handler functions

- handleHangmanSpan(span): read trimmed span.innerText, skip if empty or in hangmanSeen, add to set, call sendToServer(text) and showBadge(span, result) on reply.
- scanHangmanExisting(): query #chatBox > p and call handler for each.

- setupHangmanObserver(): find #chatBox container, create MutationObserver watching {childList:true, subtree:true}, on added call handler, call scanHangmanExisting().

Step 21: Implement Skribbl.io handler functions

- handleSkribblSpan(span): trimmed text, skip duplicates, extract message body after first colon (if present) with text.includes(":") ? text.split(":",2)[1].trim() : text, call sendToServer(message) and show badge.
- scanSkribblExisting(): select #game-chat > div.chat-content > p > span and call handler.
- setupSkribblObserver(): observe #game-chat > div.chat-content with MutationObserver; for added nodes find p > span or node.querySelector("span") and call handler; call scanSkribblExisting().

Step 22: Implement wait-and-initialize polling loop in content script

- Set maxRetries = 20, tries = 0, interval timer every 500ms: check DOM for hangman and skribbl containers, call respective setup*Observer() when found, clear interval if either found or tries >= maxRetries.

Step 23: Avoid duplicate UI clutter and performance issues

- Inside showBadge, set dataset.phishChecked so repeated observers or reflows do not duplicate badges.
- Use Sets keyed by message text to reduce server calls. Consider using message hashes instead of raw text for long messages.

Step 24: Error handling and status checking on the client

- If sendToServer returns null or { error: "Model not trained yet" }, optionally show an unobtrusive indicator (or skip) and avoid repeated retries for the same message.

Step 25: Portability and configuration best practices

- Avoid hard-coded DATASETS_DIR — use environment variable or relative paths; provide CLI/config to specify dataset location.

Step 26: Data integrity and label mapping checks

- When loading CSVs validate expected column names chat_message and category.
- Log or raise errors when unexpected labels or missing columns are encountered.

Step 27: Training considerations and resource limits

- ML uses max_iter=30; for larger datasets increase max_iter or use early stopping.

- Limit max_features in TF-IDF to control memory. Persist trained model to disk if startup training is expensive.

Step 28: Explainability and outputs for UI

- Expose keywords and heuristics in the Flask response so the extension can display more context (e.g., "contains URL", "all-caps ratio high").

Step 29: Security and deployment notes

- Do not expose local Flask server publicly.
- If deploying, secure the API with authentication, rate limiting, and HTTPS.
- Sanitize logs to avoid leaking messages.

Step 30: Testing and validation steps

- Unit-test extract_features, load_datasets, and predict with known inputs.
- Manually test extension against sample chat messages and use /status endpoint to ensure model readiness before heavy client use.

IMPLEMENTATION DETAILS

FILESYSTEM

Manifest.json	**Declares MV3 extension, permissions, injects content.js**
content.js	**Monitors chat, sends messages to server, shows badges**
chat_server.py	**Flask app, handles /detect and /status, starts training thread**
phishing_detector.py	**Loads datasets, trains model, computes predictions**

```
PhishBait
| - Datasets
        | - spam1.csv
        | - spam2.csv
        | - spam3.csv
| - Extension
        | - content.js
        | - manifest.json
| - flask_app.py
| - Hangman
        | - node_modules (directory)
        | - public
                | - client.js
                | - index.html
        | - package-lock.json
        | - package.json
        | - server.js
| - phish_detector.py
```

IMPLEMENTATION DETAILS

DATASETS

- **SMS Phishing Dataset for Machine Learning and Pattern Recognition from Mendeley**

The dataset comprises a class distribution of 489 spam, 638 smishing, and 4,844 ham messages. It provides both raw message content and pre-extracted linguistic and structural attributes derived from malicious texts.

- **Kaggle SMS Spam Collection (Text Classification)**

The dataset exhibits a class distribution of 2,154 spam messages and 3,817 not spam messages. The spam category includes a variety of phishing, smishing, and other malicious messages.

- **Custom-Generated In-Game Chat Dataset**

The dataset exhibits a class distribution of 2,711 spam messages and 7,289 not spam messages. The spam category includes messages containing phishing links, deceptive reward offers.The not spam category encompasses legitimate in-game conversations, including casual player interactions, team coordination, and general gameplay discussions. This dataset effectively captures the linguistic style of gaming environments.

IMPLEMENTATION DETAILS

METHODOLOGY

DATA PREPROCESSING:

- Three datasets were used: two publicly available SMS phishing datasets and one custom-generated in-game chat dataset.
- Data cleaning involved removing duplicates, handling missing values, and normalizing text.
- Messages were labeled as spam or not_spam, and the datasets were preprocessed to extract only relevant fields (chat_message and category).

FEATURE ENGINEERING:

- To capture message characteristics, features such as URL presence, keyword hits, punctuation ratio, and all-caps ratio were extracted.
- In addition, the raw text of chat messages was retained for input to the machine learning model.

TEXT PROCESSING:

- Since machine learning models require numerical input, TF-IDF vectorization was applied to convert chat messages into numerical feature vectors.
- The TF-IDF vectors capture the semantic importance of words within messages, enabling the model to differentiate phishing patterns from normal chat.

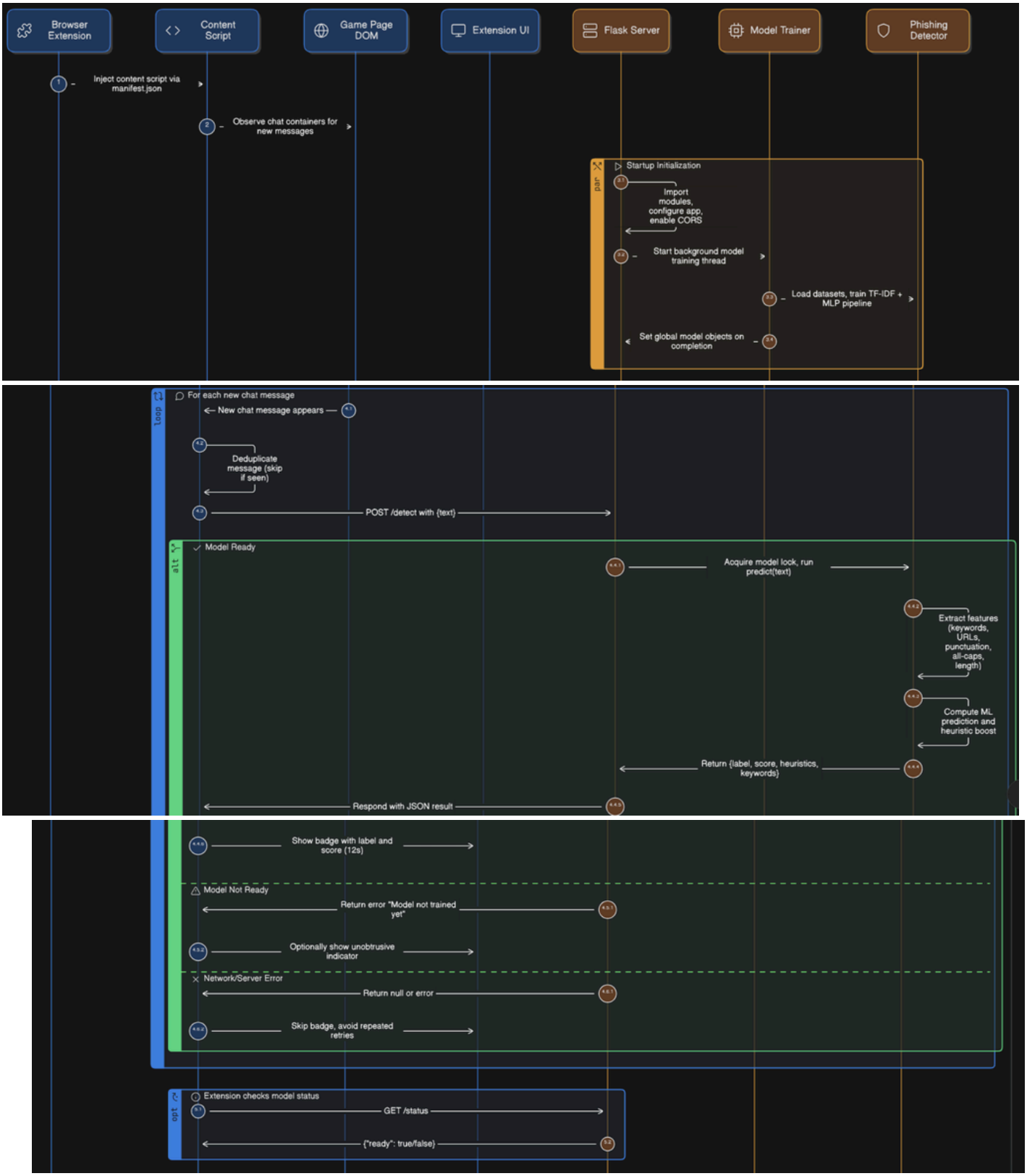
MODEL TRAINING:

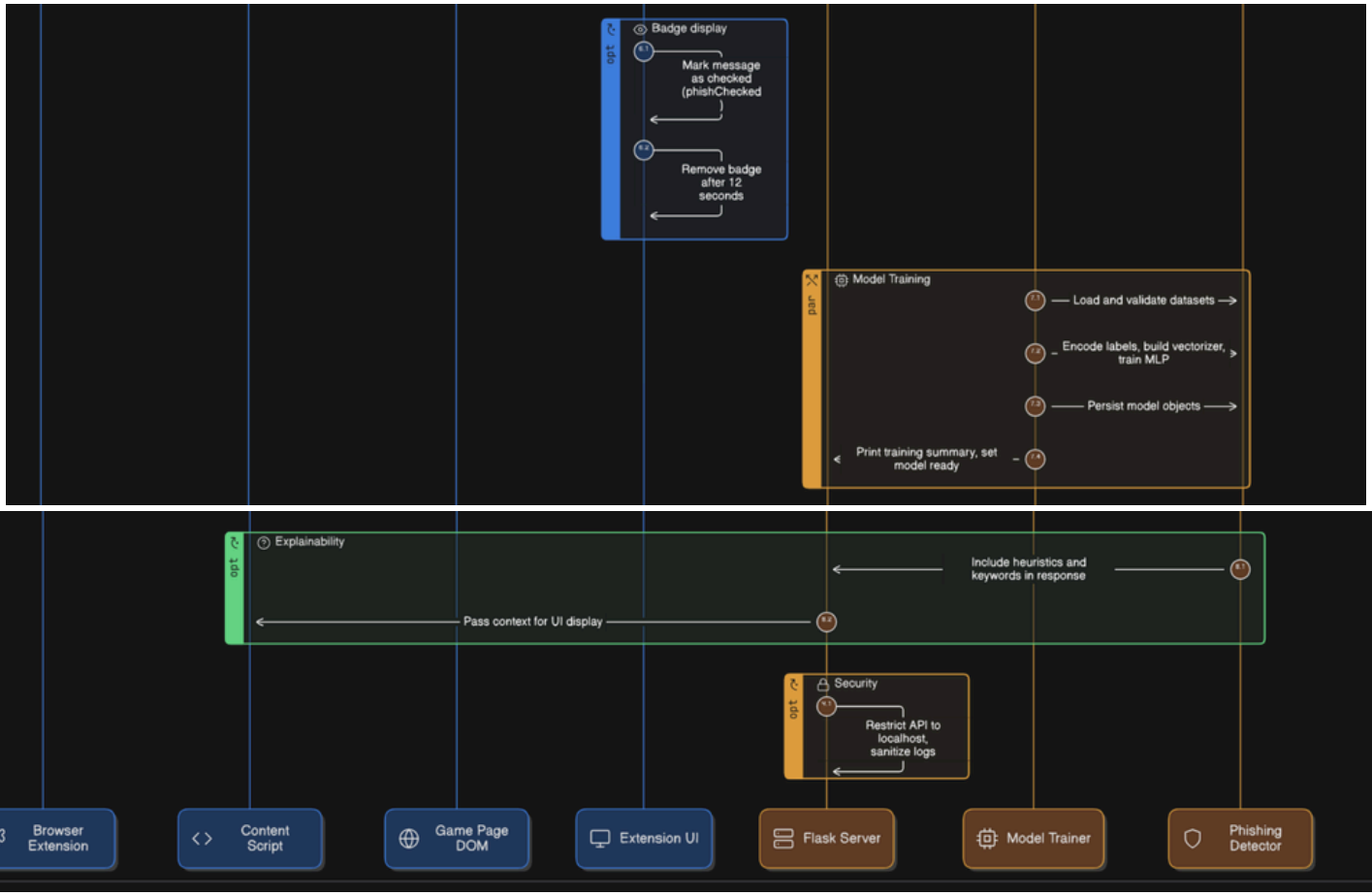
- The Multi-Layer Perceptron (MLP) classifier was trained using the TF-IDF features combined with heuristic features.
- The combined approach allows the model to leverage both statistical patterns in text and rule-based signals for accurate detection.

EVALUATION:

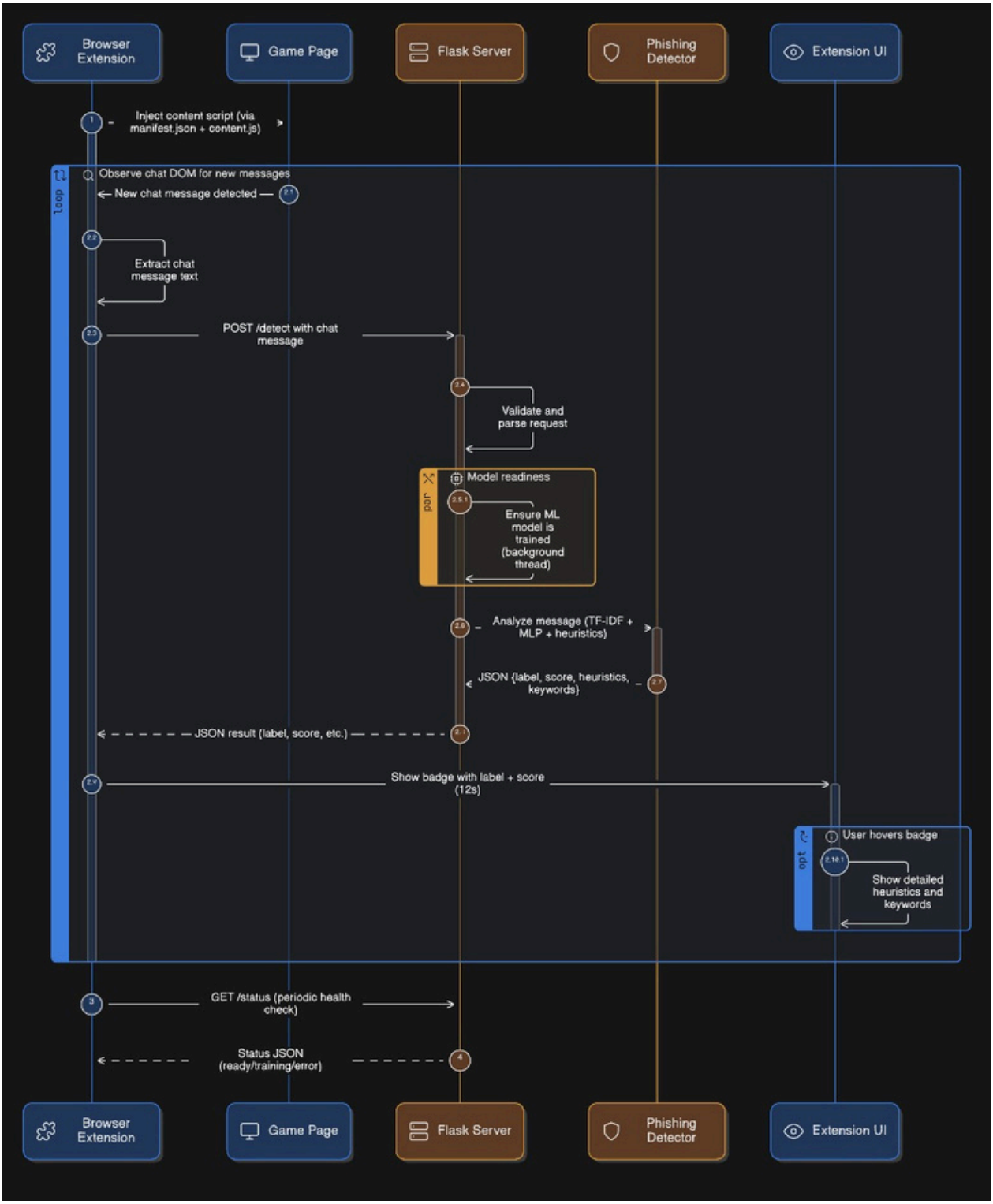
- The model was evaluated using accuracy on a held-out test set.
- Heuristic contributions were analyzed to understand which features contributed most to phishing detection.

FLOWCHART


Browser Extension
Content Script
Game Page DOM
Extension UI
Flask Server
Model Trainer
Phishing Detector
1 Inject content script via manifest.json
2 Observe chat containers for new messages
par
Startup Initialization
2.1 Import modules, configure app, enable CORS
2.6 Start background model training thread
2.7 Load datasets, train TF-IDF + MLP pipeline
2.8 Set global model objects on completion
loop
For each new chat message
3.1 New chat message appears
4.2 Deduplicate message (skip if seen)
4.3 POST /detect with {text}
alt
Model Ready
4.4.1 Acquire model lock, run predict(text)
4.4.2 Extract features (keywords, URLs, punctuation, all-caps, length)
4.4.3 Compute ML prediction and heuristic boost
4.4.4 Return {label, score, heuristics, keywords}
4.4.5 Respond with JSON result
4.4.6 Show badge with label and score (12s)
Model Not Ready
4.5.1 Return error "Model not trained yet"
4.5.2 Optionally show unobtrusive indicator
Network/Server Error
4.6.1 Return null or error
4.6.2 Skip badge, avoid repeated retries
opt
Extension checks model status
5.1 GET /status
5.2 {"ready": true/false}


opt
Badge display
Mark message as checked (phishChecked)
Remove badge after 12 seconds
par
Model Training
Load and validate datasets
Encode labels, build vectorizer, train MLP
Persist model objects
Print training summary, set model ready
opt
Explainability
Include heuristics and keywords in response
Pass context for UI display
opt
Security
Restrict API to localhost, sanitize logs
Browser Extension
Content Script
Game Page DOM
Extension UI
Flask Server
Model Trainer
Phishing Detector

ARCHITECTURE DIAGRAM



NOVELTY

- **Real-Time In-Game Chat Detection:** PhishBait analyzes messages directly within live game environments, providing instant feedback to players as they receive potentially malicious content.

- **Hybrid Detection Approach:** Combines machine learning (TF-IDF + MLP) with heuristic rules (keywords, URLs, punctuation patterns, all-caps ratio) for higher accuracy and explainability.

- **Custom In-Game Dataset:** Includes a purpose-built dataset reflecting gaming chat behavior, slang, abbreviations, and social interactions, which improves detection of context-specific phishing and scam messages.

- **Lightweight Browser Integration:** Runs entirely as a browser extension, requiring no server-side interaction for UI updates, and displays transient badges in real-time, minimizing disruption to gameplay.

RESULTS

The system's active learning approach, combined with MLP model and TF-IDF feature extraction, allowed it to adapt to informal and dynamic game chat language, significantly improving detection rates compared to traditional keyword-based filters. During testing, PhishBait consistently flagged phishing attempts with high confidence while minimizing false positives, ensuring that safe messages were not incorrectly classified.

Overall, the results indicate that the proposed system is an effective, real-time solution for enhancing cybersecurity in online multiplayer games, protecting players from scams and malicious interactions, and fostering a safe gaming environment.

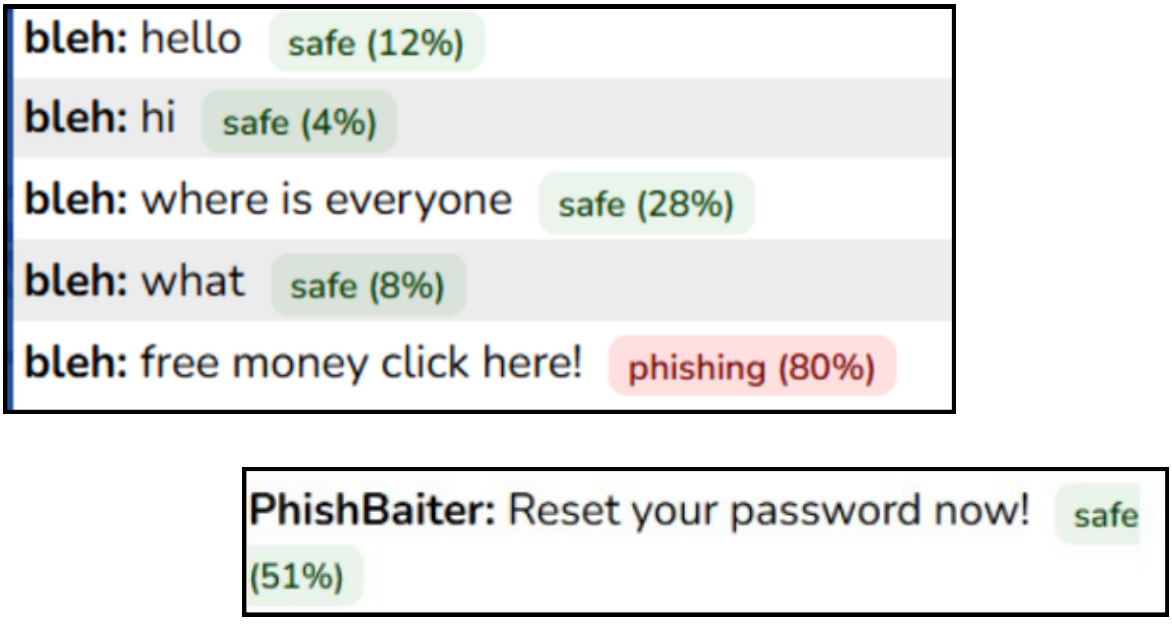


Fig. 1: Coloured Badges Indicating Probability Scores

Fig. 2: Chat Messages Classified As Phishing

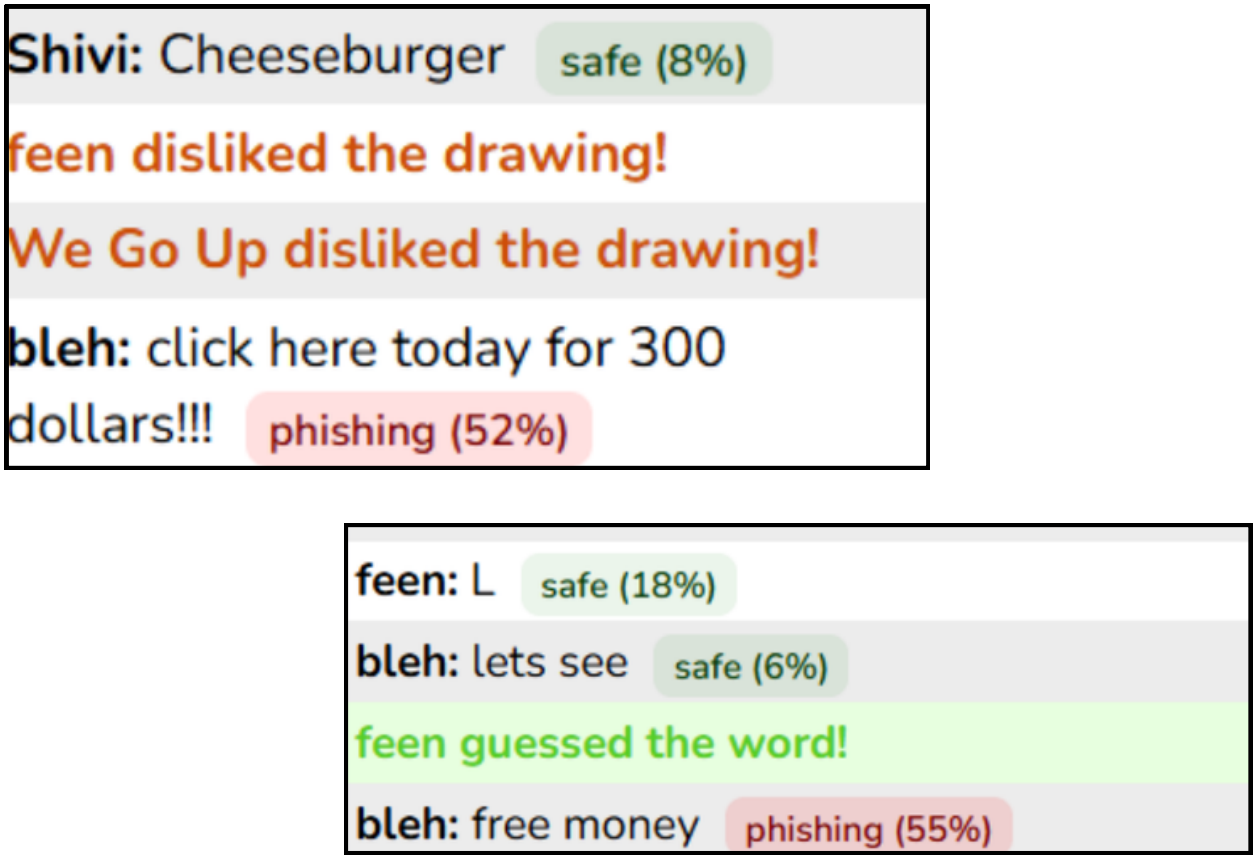
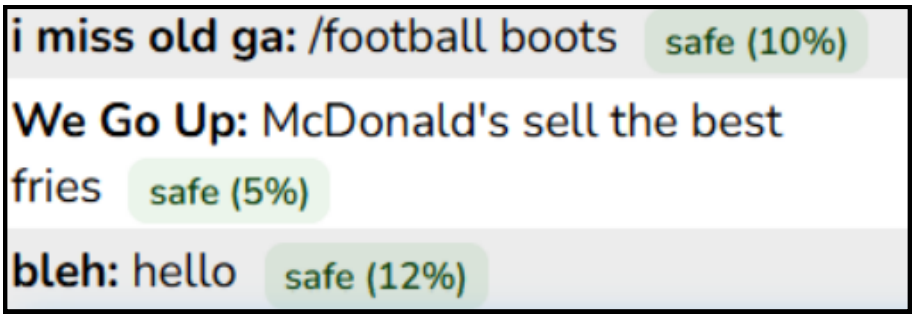


Fig. 3: Chat Messages Classified As Safe



PROJECT CODE

Extension Files:

- manifest.json
- content.js

Main Program:

- flask_app.py
- phish_detector.py

manifest.json

```
{
 "manifest_version": 3,
 "name": "PhishBait",
 "version": "1.0",
 "description": "Detects and warns about potential phishing attempts on online games",
 "permissions": ["storage", "activeTab", "scripting"],
 "content_scripts": [
  {
   "matches": [
    "*://skribbl.io/*",
    "http://localhost:3000/*"
   ],
   "js": ["content.js"]
  }
 ]
}
```

content.js

```
(function() {
  const SERVER_URL = "http://127.0.0.1:5000/detect";

  async function sendToServer(text) {
    try {
      const resp = await fetch(SERVER_URL, {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ text })
      });
      if (!resp.ok) return null;
      return await resp.json();
    } catch (e) {
      return null;
    }
  }

  function showBadge(targetSpan, result) {
    if (!targetSpan || !result) return;
    if (targetSpan.dataset.phishChecked) return;
    targetSpan.dataset.phishChecked = "true";

 const badge = document.createElement("span");
 badge.className = "phish-badge";
 badge.textContent = result.label ? `${result.label}
(${Math.round(result.score * 100)}%)` : "unknown";
```

```
badge.style.marginLeft = "8px";
badge.style.padding = "2px 6px";
badge.style.borderRadius = "6px";
badge.style.fontSize = "11px";
badge.style.fontWeight = "600";
badge.style.verticalAlign = "middle";

if (result.label === "phishing") {
badge.style.background = "rgba(255, 0, 0, 0.12)";
badge.style.color = "#7a0303";
} else {
badge.style.background = "rgba(0, 128, 0, 0.08)";
badge.style.color = "#083b09";
}

targetSpan.appendChild(badge);
setTimeout(() => badge.remove(), 12000);
}

const hangmanSeen = new Set();
const hangmanContainerSelector = "#chatBox";

async function handleHangmanSpan(span) {
const text = span.innerText?.trim();
if (!text || hangmanSeen.has(text)) return;
hangmanSeen.add(text);
const result = await sendToServer(text);
if (result) showBadge(span, result);
}
```

```
function scanHangmanExisting() {
const spans = document.querySelectorAll("#chatBox > p");
spans.forEach(handleHangmanSpan);
}

function setupHangmanObserver() {
const container =
document.querySelector(hangmanContainerSelector);
if (!container) return;

const observer = new MutationObserver(mutations => {
mutations.forEach(m => {
m.addedNodes.forEach(node => {
if (!(node instanceof HTMLElement)) return;
if (node.matches("p")) handleHangmanSpan(node);
});
});
});

observer.observe(container, { childList: true, subtree: true
});
scanHangmanExisting();
console.log("Hangman observer set up.");
}

const skribblSeen = new Set();
const chatSelector = "#game-chat > div.chat-content > p >
span";
const chatContainerSelector = "#game-chat > div.chat-
content";
```

```
 async function handleSkribblSpan(span) {
 const text = span.innerText?.trim();
 if (!text || skribblSeen.has(text)) return;
 skribblSeen.add(text);
 const message = text.includes(":") ? text.split(":", 2)
[1].trim() : text;
 const result = await sendToServer(message);
 if (result) showBadge(span, result);
 }

 function scanSkribblExisting() {

document.querySelectorAll(chatSelector).forEach(handleSkri
bblSpan);
 }

 function setupSkribblObserver() {
 const container =
document.querySelector(chatContainerSelector);
 if (!container) return;

 const observer = new MutationObserver(mutations => {
 mutations.forEach(m => {
 m.addedNodes.forEach(node => {
 if (!(node instanceof HTMLElement)) return;
 const span = node.matches("p > span") ? node :
node.querySelector("span");
 if (span) handleSkribblSpan(span);
 });
 });
 });
```

```
 observer.observe(container, { childList: true, subtree: true
});
 scanSkribblExisting();
 }

 const maxRetries = 20;
 let tries = 0;
 const interval = setInterval(() => {
 const hangmanReady =
document.querySelector(hangmanContainerSelector);
 const skribblReady =
document.querySelector(chatContainerSelector);
 if (hangmanReady) setupHangmanObserver();
 if (skribblReady) setupSkribblObserver();
 if (hangmanReady || skribblReady) clearInterval(interval);
 tries += 1;
 if (tries >= maxRetries) clearInterval(interval);
 }, 500);
})();
```

flask_app.py

```
from flask import Flask, request, jsonify
from flask_cors import CORS
from phish_detector import predict, start_training_thread,
_model
from sklearn.exceptions import NotFittedError

HOST = "127.0.0.1"
PORT = 5000

app = Flask(__name__)
CORS(app)

@app.route("/detect", methods=["POST"])
def detect():
    payload = request.get_json(force=True, silent=True)
    if not payload:
        return jsonify({"error": "Invalid JSON"}), 400

    text = payload.get("text", "").strip()
    if not text:
        return jsonify({"error": "Empty text"}), 400

    try:
        result = predict(text)
        return jsonify(result)
    except NotFittedError:
        return jsonify({"error": "Model not trained yet"}), 503
    except Exception as e:
        return jsonify({"error": "Internal server error",
"details": str(e)}), 500
```

```
@app.route("/status", methods=["GET"])
def status():
 ready = _model is not None
 return jsonify({"ready": ready})

if __name__ == "__main__":
 start_training_thread()
 print(f"[SERVER] Flask app running on http://{HOST}:
{PORT}")
 app.run(host=HOST, port=PORT, debug=False)
```

phish_detector.py

```
import os
import re
import threading
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import LabelEncoder
from sklearn.neural_network import MLPClassifier
from sklearn.exceptions import NotFittedError

BASE_DIR = os.path.dirname(os.path.abspath(__file__))
DATASETS_DIR =
r"C:\Users\anagh\OneDrive\Desktop\abp\college\SEM
5\ai&nn\oratio\gamephisher\Datasets"
CSV_FILES = ["spam1.csv", "spam2.csv", "spam3.csv"]
SUSPICIOUS_KEYWORDS =
["click","here","free","win","urgent","prize","reward","verif
y","account","password","login","bank","update","confirm",
"activate","validate","bitcoin","money"]
URL_PATTERN = re.compile(r"https?://\S+|www\.\S+")

_model = _vectorizer = _label_encoder = None
_model_lock = threading.Lock()
KEYWORD_TOP_K = 5
```

```
def extract_features(text):
 words = text.split()
 return {
 "keyword_count": sum(1 for kw in
SUSPICIOUS_KEYWORDS if kw in text.lower()),
 "has_url": int(bool(URL_PATTERN.search(text))),
 "exclamations": text.count("!"),
 "questions": text.count("?"),
 "all_caps_ratio": sum(1 for w in words if w.isupper()) /
len(words) if words else 0,
 "length": len(text)
 }


def load_datasets():
 dfs = []
 for file in CSV_FILES:
 path = os.path.join(DATASETS_DIR, file)
 if not os.path.isfile(path): continue
 df = pd.read_csv(path, usecols=
["chat_message","category"]).dropna()
 df.columns = ["text","label"]
 df["label"] =
df["label"].replace({"spam":"phishing","not_spam":"safe"})
 dfs.append(df)
 if not dfs: raise FileNotFoundError("No valid datasets
found.")
 return pd.concat(dfs, ignore_index=True)
```

```
def train_model():
 global _model, _vectorizer, _label_encoder
 df = load_datasets()
 texts, labels = df["text"].tolist(), df["label"].tolist()

 le = LabelEncoder()
 y = le.fit_transform(labels)
 vec = TfidfVectorizer(ngram_range=(1,2),
max_features=15000, stop_words="english")
 clf = MLPClassifier(hidden_layer_sizes=(128,64),
activation='relu', solver='adam', max_iter=30,
random_state=42, verbose=True)

 pipeline = make_pipeline(vec, clf)
 pipeline.fit(texts, y)

 _model, _vectorizer, _label_encoder = pipeline, vec, le
 print(f"[ML] Trained on {len(texts)} samples. Classes:
{list(le.classes_)}")

def start_training_thread():
 threading.Thread(target=lambda: train_model(),
daemon=True).start()

def extract_keywords(text, top_k=KEYWORD_TOP_K):
 if _vectorizer is None: return []
 X = _vectorizer.transform([text]).toarray()[0]
 feature_names = _vectorizer.get_feature_names_out()
 return [feature_names[i] for i in np.argsort(X)[::-1][:top_k]
if X[i]>0]
```

```
def predict(text):
 with _model_lock:
 if _model is None: raise NotFittedError("Model not ready.")
 proba = _model.predict_proba([text])[0]
 phishing_prob =
float(proba[np.where(_label_encoder.classes_=="phishing")
[0][0]])

 feats = extract_features(text)
 boost = 0.2*(feats["keyword_count"]>0) +
0.3*feats["has_url"] + 0.1*(feats["all_caps_ratio"]>0.5) +
0.1*((feats["exclamations"]>2) or (feats["questions"]>2))
 score = min(phishing_prob + boost, 1.0)
 label = "phishing" if score>=0.5 else "safe"

 return {
 "label": label,
 "score": round(score,4),
 "keywords": extract_keywords(text),
 "raw_prediction": f"phishing_prob={phishing_prob:.3f}",
 "heuristics": feats
 }
```

CONCLUSION

PhishBait offers an efficient, real-time safeguard against phishing attempts in live game chats through the seamless integration of a lightweight browser extension, a local Flask gateway, and an MLP-based detection model. Building on a keyword and TF-IDF baseline enhanced with active learning, it ensures fast and accurate message classification while maintaining smooth, user-friendly performance. By preventing malicious interactions, protecting user accounts, and promoting safer online communities, PhishBait enhances player trust and enables a more secure and enjoyable gaming experience.

REFERENCES

- https://www.videosdk.live/developer-hub/media-server/video-game-chat-guide
- https://ieeexplore.ieee.org/document/4597251
- https://developer.chrome.com/docs/extensions/get-started
- https://www.cm-alliance.com/cybersecurity-blog/phishing-scams-in-the-gaming-community

COLLABORATIVE GITHUB

- https://github.com/4n4gh4/PhishBait