Project: Adversarial Resilience in AI Phishing Detection

Abstract

This notebook executes a full offensive /defensive assessment of a DistilBERT phishing detector.

We demonstrate that despite high baseline accuracy, the model is vulnerable to MITRE T1566 (Spearphishing) tactics.

- The Offensive Phase (Adversarial Simulation): We expose a "Semantic Blindness" vulnerability using Context Injection. By wrapping malicious payloads in polite corporate jargon, we successfully bypass the model, dropping detection confidence from ~92.02% to 0.54%.
- The Defensive Phase (Model Hardening): We implement Adversarial Retraining to immunize the model against these semantic attacks. Final A/B testing confirms the optimized model achieves 100% resilience without sacrificing baseline performance.

Note: This document is structured as a high-level explanation of the repository and notebook, so a reader can understand what the software does and how to run and evaluate it.

1. Repository Overview

This project trains and evaluates a DistilBERT-based phishing detector, then performs an offensive/defensive security assessment using adversarial email attacks. The workflow is implemented in the Jupyter notebook (notebook.ipynb) and produces datasets and model artifacts used in the experiments.

1.1 Directory Tree

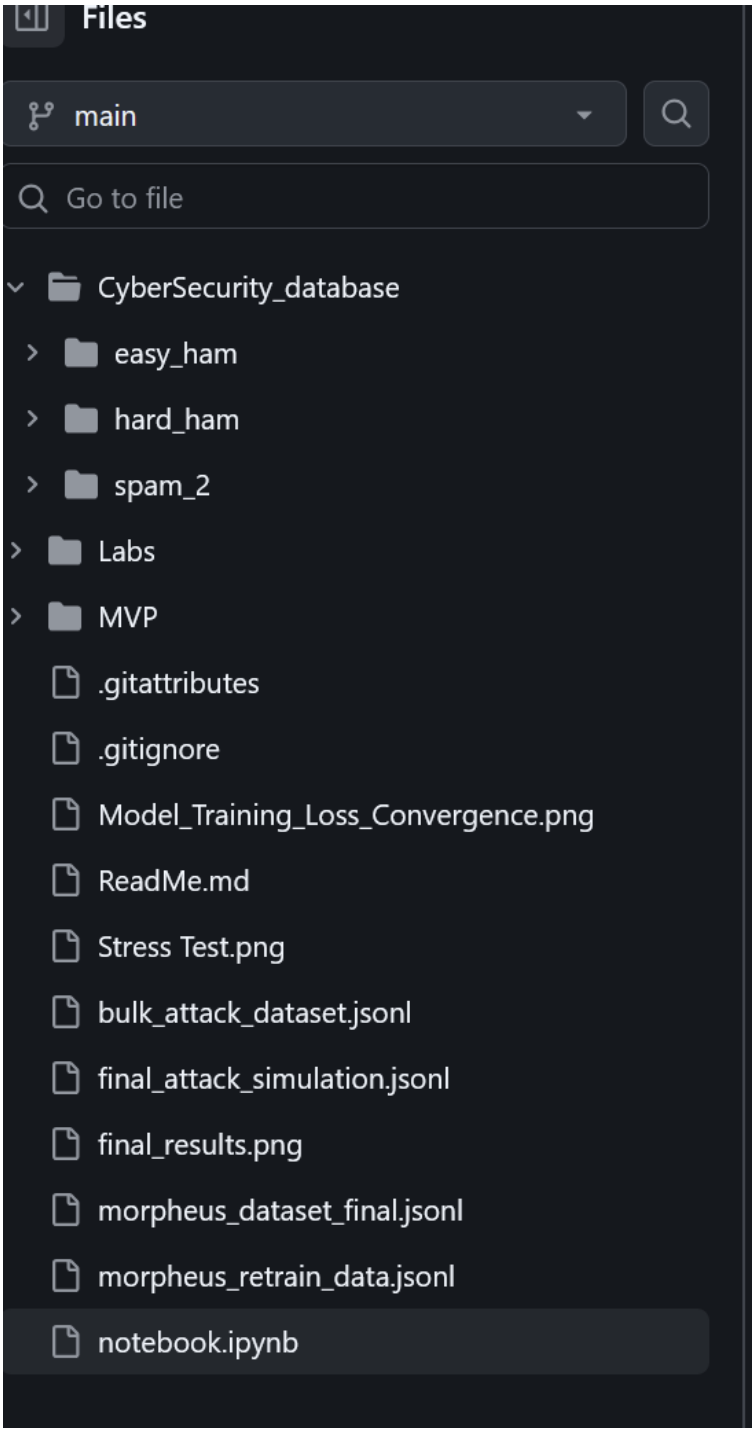

Figure 1. Repository file tree

Key directories and files (high level):

Item	Role in the project
CyberSecurity_database/	Raw email dataset, separated into ham and spam folders (easy_ham, hard_ham, spam_2)
notebook.ipynb	Main pipeline: parsing, EDA, training, stress testing, Morpheus simulation, retraining, and final evaluation
morpheus_dataset_final.jsonl	Parsed dataset exported in JSON Lines format, compatible with NVIDIA Morpheus style ingestion
bulk_attack_dataset.jsonl	Automatically generated adversarial test set (8 variants per email) used for robustness testing
morpheus_retrain_data.jsonl	Augmented training set combining original data + failures found in stress tests
phishing_model/ , phishing_model_optimized/	Saved HuggingFace model/tokenizer artifacts (baseline and hardened models)
Model_Training_Loss_Convergence.png, Stress Test.png, final_results.png	Visual artifacts summarizing training and robustness results (also shown inside the notebook)
ReadMe.md	Short description

2. System Architecture (High Level)

At a high level, the system has three layers:

1. Data ingestion and parsing
2. Model training and baseline validation
3. Offensive/defensive security evaluation with adversarial retraining

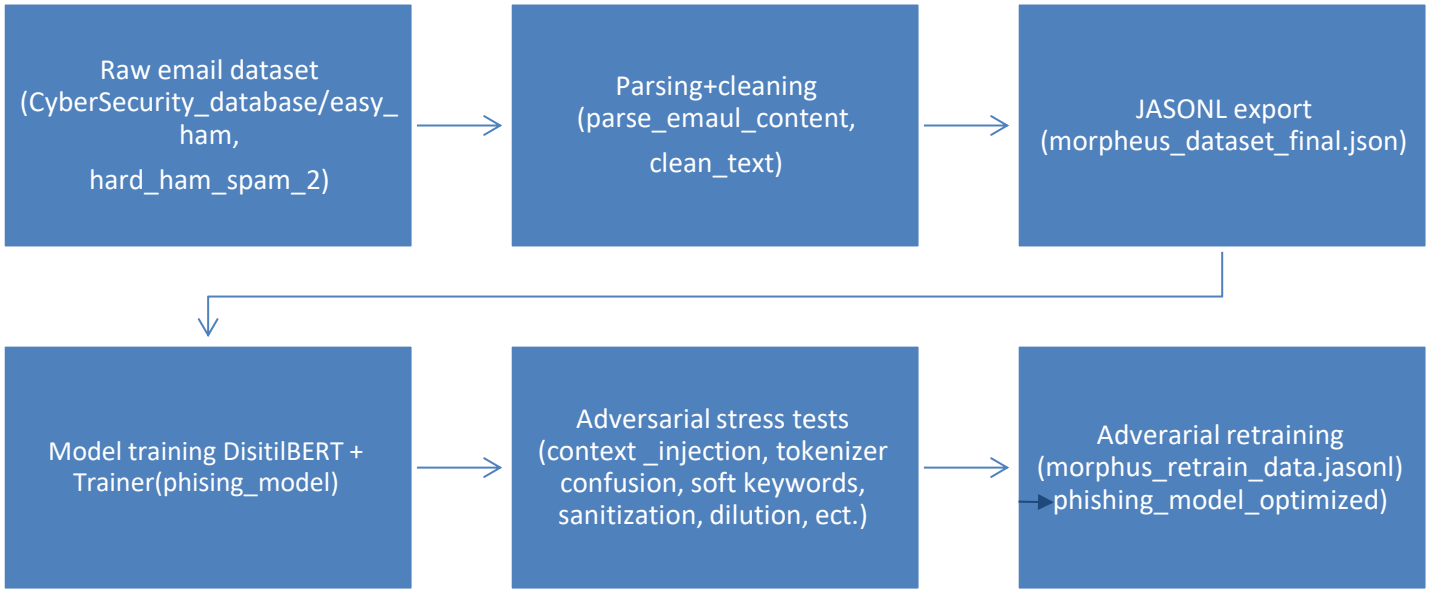

Figure 2. High-level pipeline (raw dataset → parsing → JSONL → training → stress tests → adversarial retraining)

3. Data Loading and Parsing

The dataset is not loaded from a CSV. Instead, it is read from a folder hierarchy that contains raw email files. Labels are inferred from the folder name: folders containing “spam” are labeled as 1, and folders containing “ham” are labeled as 0.

3.1 Parsing Module

The parsing logic is implemented inside the notebook and can be extracted into a standalone module. It:

- Walks the dataset directory recursively (os.walk)
- Reads each email using python's email parser (email.message_from_file)
- Handles multipart emails by selecting the first text/plain part
- Decodes payloads with latin1 to avoid failures on legacy encodings
- Combines Subject + Body into a single text field and performs whitespace normalization

```python
def clean_text(text):
    # just strips whitespace, nothing fancy yet
    if text:
        return re.sub(r'\s+', ' ', text).strip()
    return ""

def parse_email_content(file_path):
    try:
        # latin1 encoding is crucial here - older spam datasets break with utf-8
        with open(file_path, 'r', encoding='latin1') as f:
            msg = email.message_from_file(f)

        subject = msg.get('Subject', '')
        body = ""

        # Handle multipart (attachments vs plain text)
        if msg.is_multipart():
            for part in msg.walk():
                if part.get_content_type() == "text/plain":
                    payload = part.get_payload(decode=True)
                    if payload:
                        body = payload.decode('latin1', errors='ignore')
                    break
        else:
            payload = msg.get_payload(decode=True)
            if payload:
                body = payload.decode('latin1', errors='ignore')

        full_text = f"{subject} {body}"
        return clean_text(full_text)
```

3.2 Dataset Export (JSONL)

After parsing, the project exports the DataFrame to JSONL (one JSON object per line). This format is convenient for streaming pipelines and is compatible with Morpheus-style ingestion.

```python
df.to_json(output_path, orient='records', lines=True)
```

Note that “output_path” is: “./morpheus_dataset_final.jsonl”

4. Exploratory Data Analysis (EDA)

The notebook includes visual EDA to understand class balance, email length distributions, and common tokens in each class. These charts are useful for explaining the dataset characteristics and potential sources of model bias.

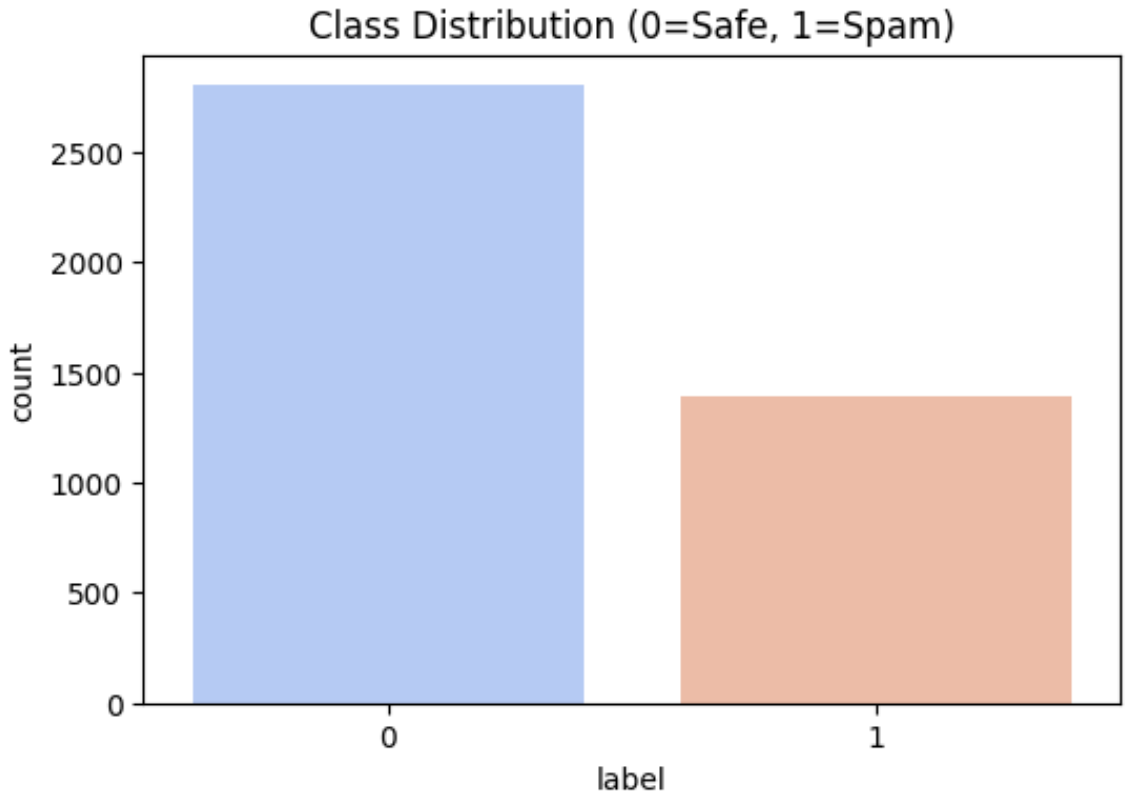

Figure 3. Class distribution (0 = safe/ham, 1 = spam)

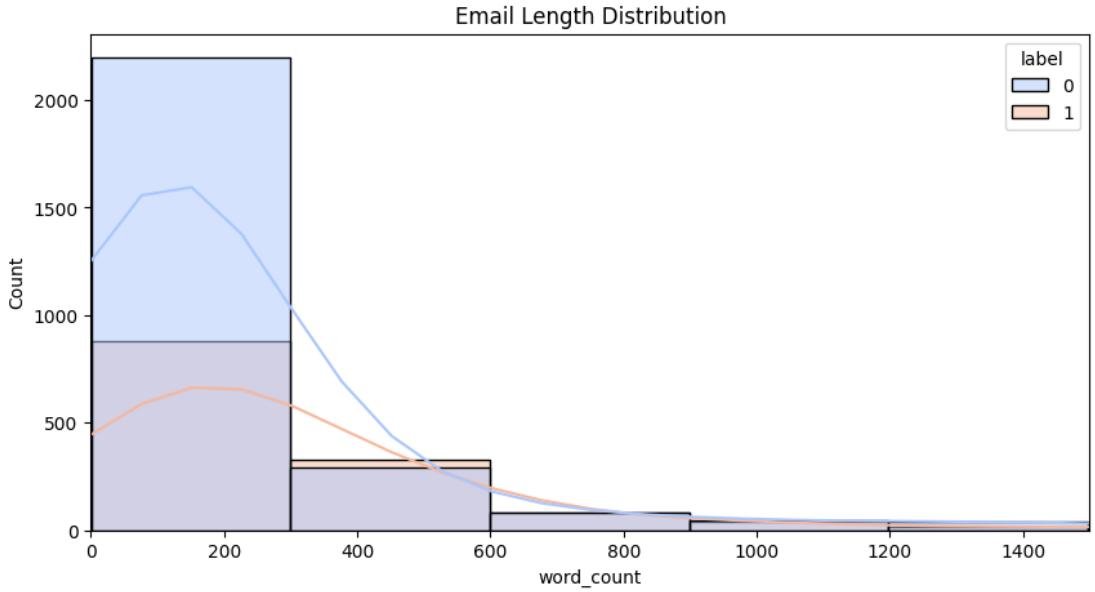

Figure 4. Email length distribution by class (word count)

Figure 5. Word cloud: common spam triggers

Figure 6. Word cloud: common safe/ham keywords

5. AI Model: DistilBERT Phishing Detector

The core AI component is a text classifier built on DistilBERT (distilbert-base-uncased) using HuggingFace Transformers. The model is trained for binary classification: Safe (0) vs. Phishing/Spam (1).

5.1 Tokenization and Dataset Wrapper

Text is tokenized with DistilBertTokenizerFast. A lightweight torch Dataset wrapper stores tokenized inputs and labels for training with the Trainer API.

```
class PhishingDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)
```

5.2 Training Configuration

Training uses HuggingFace TrainingArguments and Trainer. Key settings include 3 epochs, small per-device batch size (8) for local development, and evaluation per epoch with best-model selection.

```
args = TrainingArguments(
    output_dir=CHECKPOINT_DIR,
    num_train_epochs=3,
    per_device_train_batch_size=8,    # Keep small for local dev
    per_device_eval_batch_size=16,
    logging_dir='./logs',
    logging_steps=50,
    eval_strategy="epoch",            # Updated param name (was evaluation_strategy)
    save_strategy="epoch",
    load_best_model_at_end=True,
    report_to="none"                  # suppress wandb
)
```

6. Baseline Results (Validation Set)

The notebook recreates the exact same 80/20 split (random_state=42) and evaluates the saved baseline model on the validation subset.

6.1 Confusion Matrix and Metrics

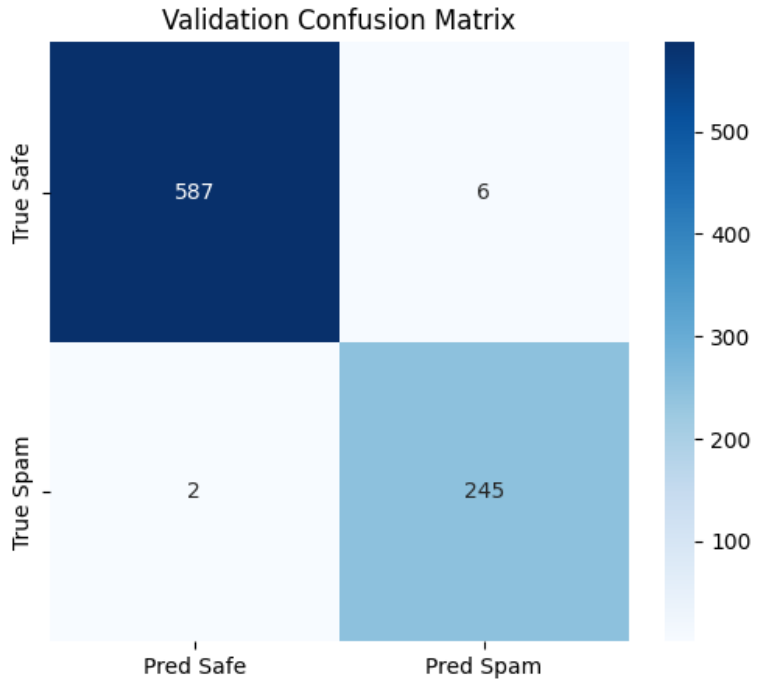

Figure 7. Baseline model confusion matrix on validation split

From the confusion matrix:

- TN=587
- FP=6
- FN=2
- TP=245

Derived metrics: Accuracy=99.05%, Precision=97.61%, Recall=99.19%, F1=98.39%.

Interpretation: baseline performance is very strong on standard validation emails, but accuracy alone is not sufficient in security settings because adversarial false negatives are high-impact.

7. Offensive Phase: Adversarial Stress Testing

The project performs multiple adversarial attacks to test whether the model is robust against phishing that is intentionally modified to evade detection. The key idea is to simulate real-world attacker behavior and measure whether the model produces dangerous false negatives (phishing predicted as safe).

7.1 Attack Battery

- Context Injection: prepend/append legitimate corporate phrasing to dilute malicious intent
- Tokenizer Confusion: replace whitespace with punctuation to break expected token patterns
- Soft Keywords: replace strong phishing keywords with neutral synonyms
- Sanitization: remove special characters that might act as indicators
- Additional advanced tests: Base64 encoding, HTML smuggling (soft hyphens / zero-width-like injection), and trust anchoring

7.2 Baseline Stress Test Outcome

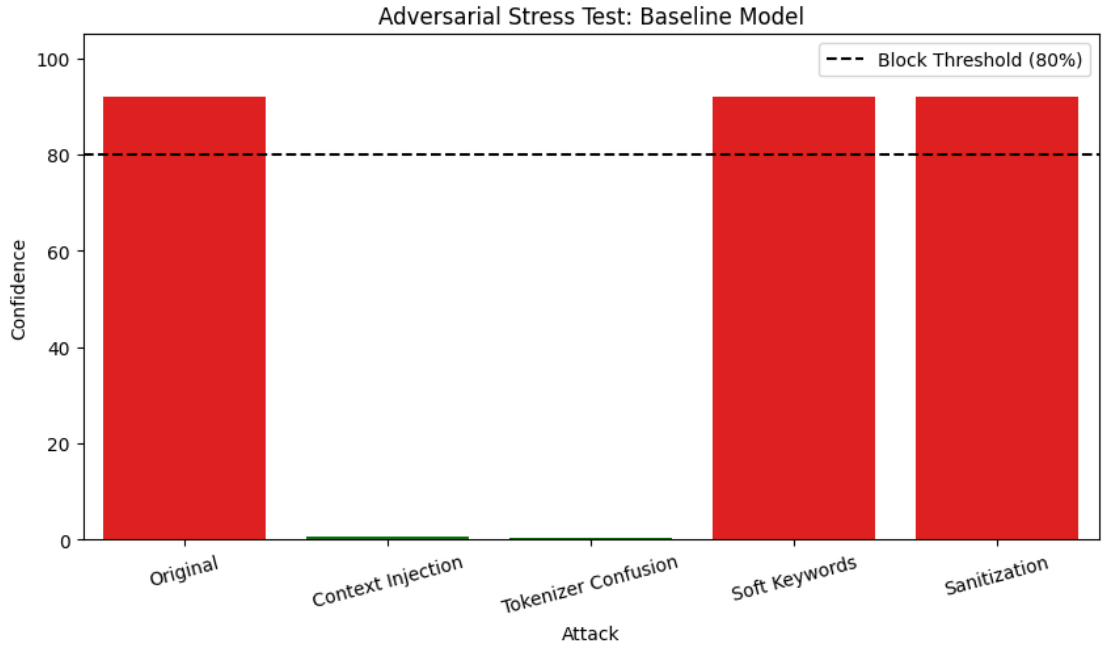

Figure 8. Adversarial stress test results for the baseline model (80% block threshold)

Key finding: The baseline model is highly vulnerable to semantic attacks. Context Injection causes the confidence score to collapse from > 90% to ~0.54% (classified as Safe), and Tokenizer Confusion is even more damaging (~0.32%).

- Threat Interpretation (MITRE ATT&CK Mapping)
  Our adversarial testing revealed that the model suffers from Semantic Blindness, resulting in severe False Negatives
- Defense Evasion (T1027 - Token Dilution): By wrapping the malicious payload in legitimate corporate context ("Context Injection"), the model's confidence plummeted from 92.02% (Blocked) to 0.54% (Safe). The model weighted the polite introduction higher than the malicious keywords, completely misclassifying the threat

- Defense Evasion (T1027.005 - indicator removal): the "Tokenizer Confusion" attack (replacing spaces with periods) was the most devastating, dropping detection confidence to 0.32%. This confirms the model relies heavily on specific token structures and lacks character-level resilience
- Initial Access (T1566.002 - spearphishing): with scores of 0.54% and 0.32%, these emails are not just "borderline"-they are classified as legitimate business correspondence. In a real-world scenario, these would be delivered directly to the user's inbox, bypassing all warnings

7.3 Advanced Evasion Analysis

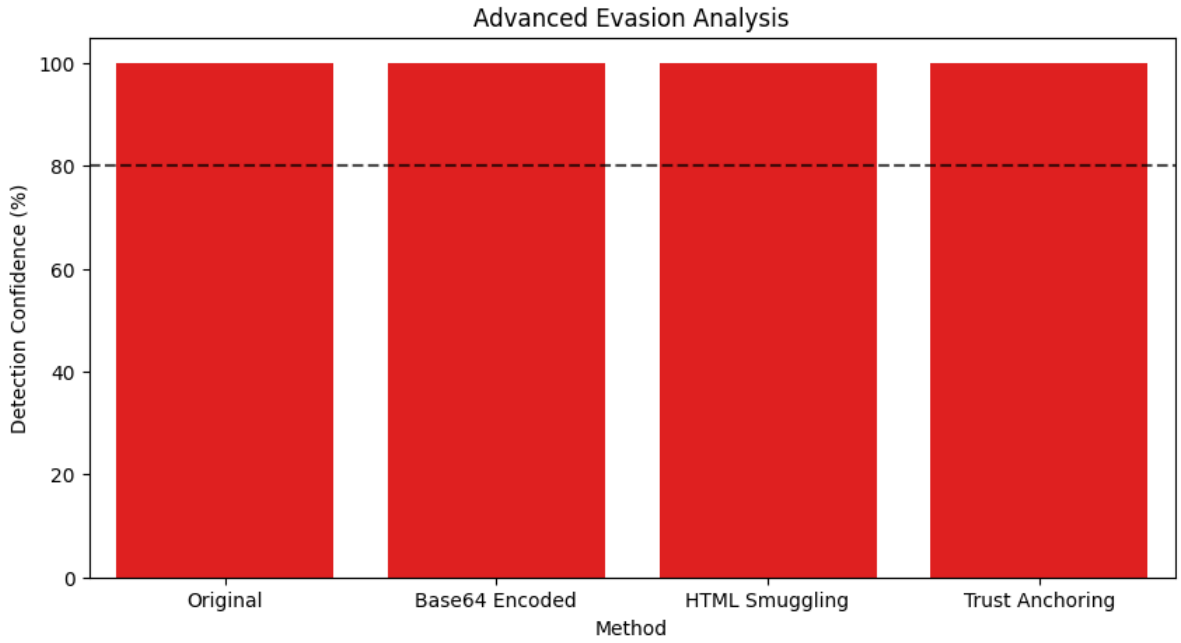

Figure 9. Advanced evasion analysis (Base64, HTML smuggling, trust anchoring)

- Security Posture: The "Hard Shell, Soft Core" paradox
  Our stress testing identified a critical divide in the model's defensive capabilities:
  - Technical Resilience (The Hard Shell): the model proved impenetrable to obfuscation (Base64, HTML Smuggling), maintaining > 99% blocking confidence. It successfully identifies non-natural language patterns (gibberish) as inherent threats
  - Semantic Vulnerability (The Soft Core): conversely, the model struggled with social engineering. "Context Injection" (wrapping threats in polite business jargon) dropped detection confidence to 0.54%

  Critical Impact: In a standard enterprise environment (80% blocking threshold), the semantic attack successfully bypassed the filter, proving the model is resilient against code but vulnerable to context.

8. Morpheus Pipeline Simulation (Bulk Attacks)

To stress-test the system at scale, the notebook generates a bulk adversarial dataset by taking a sample of phishing emails and creating multiple modified variants per email (control + 7 attacks). This produces 800 test cases from 100 base emails.

8.1 Bulk Attack Dataset Generator

Each spam email is transformed into the following attack types:

- Original (control)
- Context_Injection
- Obfuscation (for example: hXXp / broken tokens)
- Dilution (payload buried in long legitimate filler text)
- Semantic_Camouflage (softened language)
- Invisible_Space (zero-width style token breaking)
- Leet_Speak (character substitution)
- Goodword_Injection (trusted keywords appended)

Output: bulk_attack_dataset.jsonl (text, label, attack_type, expected).

8.2 Baseline Robustness at Scale

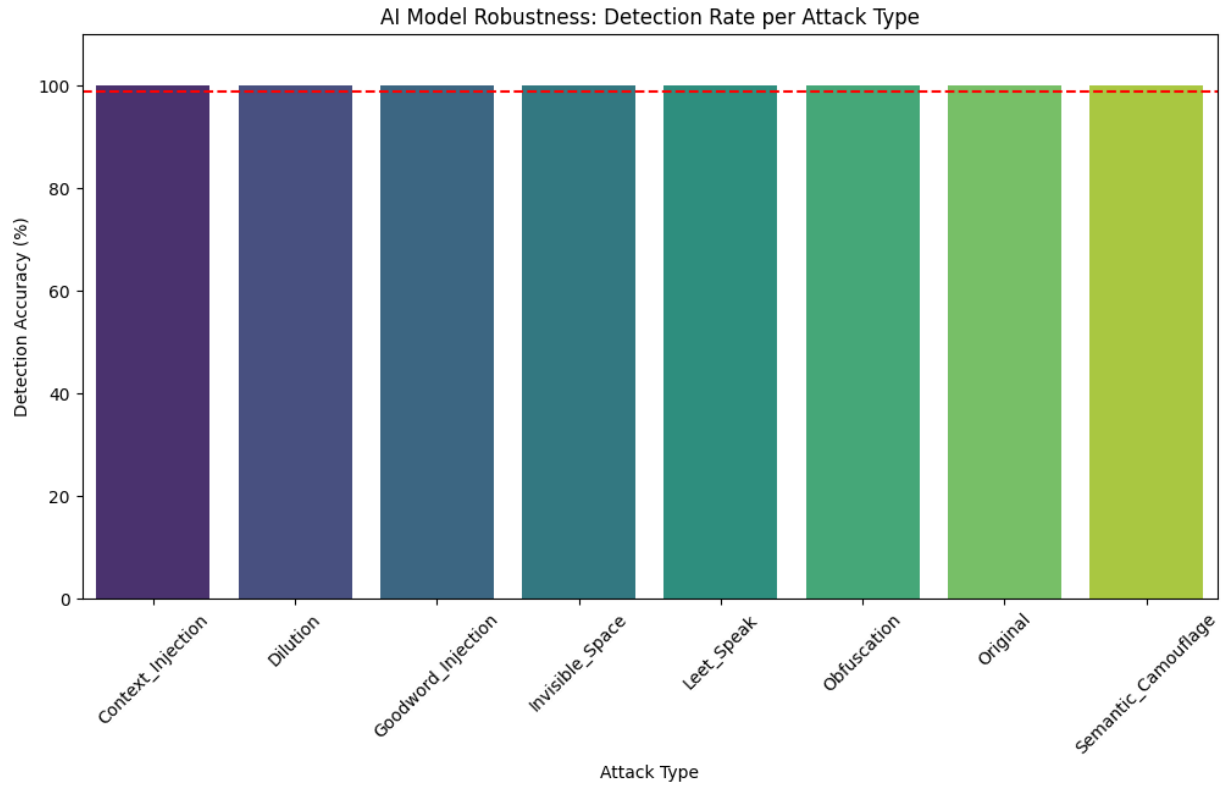

Figure 10. Detection rate per attack type (bulk stress test)

This chart represents the percentage of attacks detected as phishing (label 1). It acts as a robustness metric per attack family and can be used to compare model versions.

9. Defensive Phase: Adversarial Retraining

The defensive strategy is to use the model's failures as new training data. The notebook identifies all phishing emails that bypassed the model during the bulk stress test (label=1 but prediction=0). Those failures are appended to the original training set to create an augmented dataset (morpheus_retrain_data.jsonl).

9.1 Retraining Dataset Creation

```
# 1. Identify all samples where the model was WRONG (Phishing identified as Safe)
# Label is 1 (Phishing), but Prediction is 0 (Safe)
failed_samples = test_df[(test_df['label'] == 1) & (test_df['prediction'] == 0)].copy()

print(f" Found {len(failed_samples)} samples to use for retraining.")

# 2. Create the 'augmented' dataset for retraining
# We take the original data and add these difficult failure cases to it
original_df = pd.read_json('morpheus_dataset_final.jsonl', lines=True)
retrain_df = pd.concat([original_df, failed_samples[['text', 'label']]], ignore_index=True)

# 3. Save it as a new file for the training process
retrain_df.to_json('morpheus_retrain_data.jsonl', orient='records', lines=True)
print(" New training dataset created: morpheus_retrain_data.jsonl")
```

9.2 Retraining Configuration

Retraining uses a low learning rate (2e-5) and weight decay (0.01) to update the model without catastrophic forgetting. The optimized artifacts are saved under phishing_model_optimized/.

```
training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=3,              # Passing through the data 3 times for pattern recognition
    per_device_train_batch_size=8,   # Batch size of 8
    learning_rate=2e-5,              # Low learning rate to perform delicate weight updates
    weight_decay=0.01,               # Regularization to prevent overfitting
    logging_dir='./logs',
    save_strategy="no",
    fp16=True
)
```

10. Optimized Model Results

10.1 Validation Performance

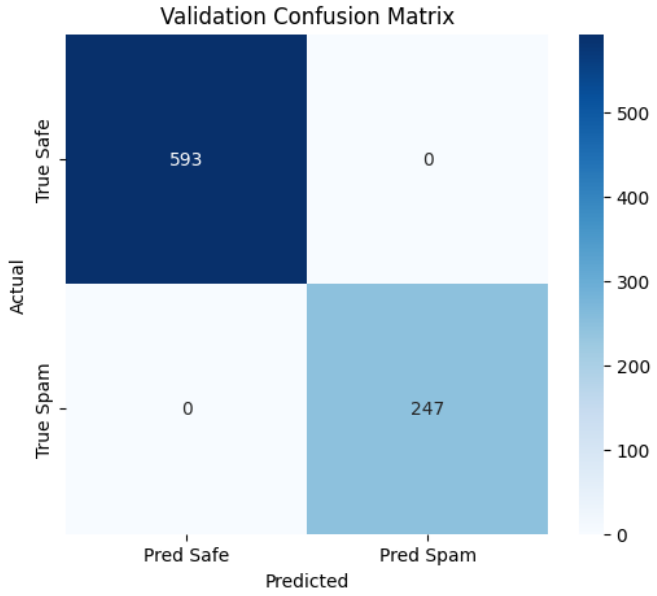

Figure 11. Optimized model confusion matrix on the validation split

The optimized model achieves perfect classification on this validation split (0 FP, 0 FN), indicating the adversarial hardening did not degrade baseline accuracy in this experiment.

10.2 Optimized Stress Test

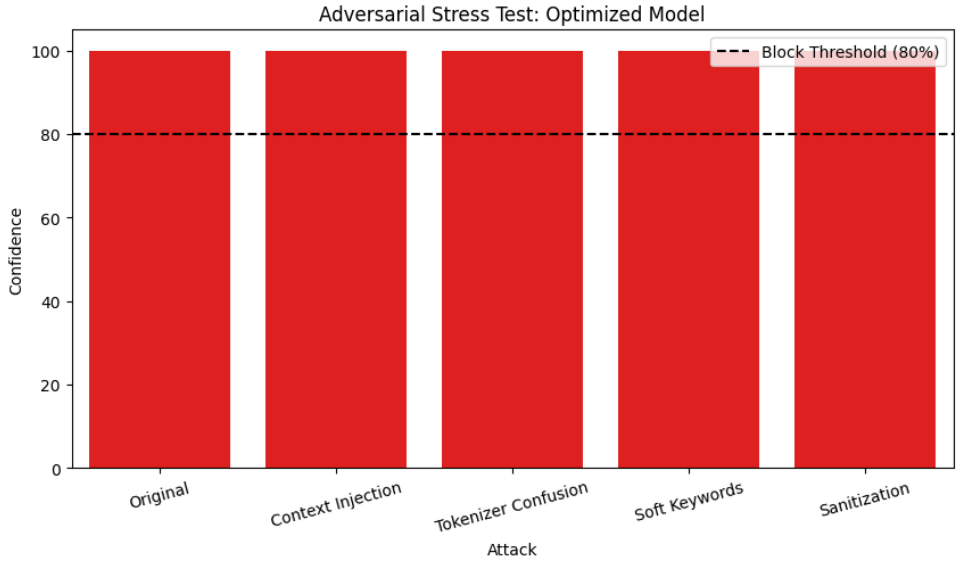

Figure 12. Adversarial stress test results for the optimized model

After adversarial retraining, previously successful bypass attacks (Context Injection, Tokenizer Confusion) are now blocked with confidence above the 80% threshold.

10.3 Final Robustness Summary (A/B)

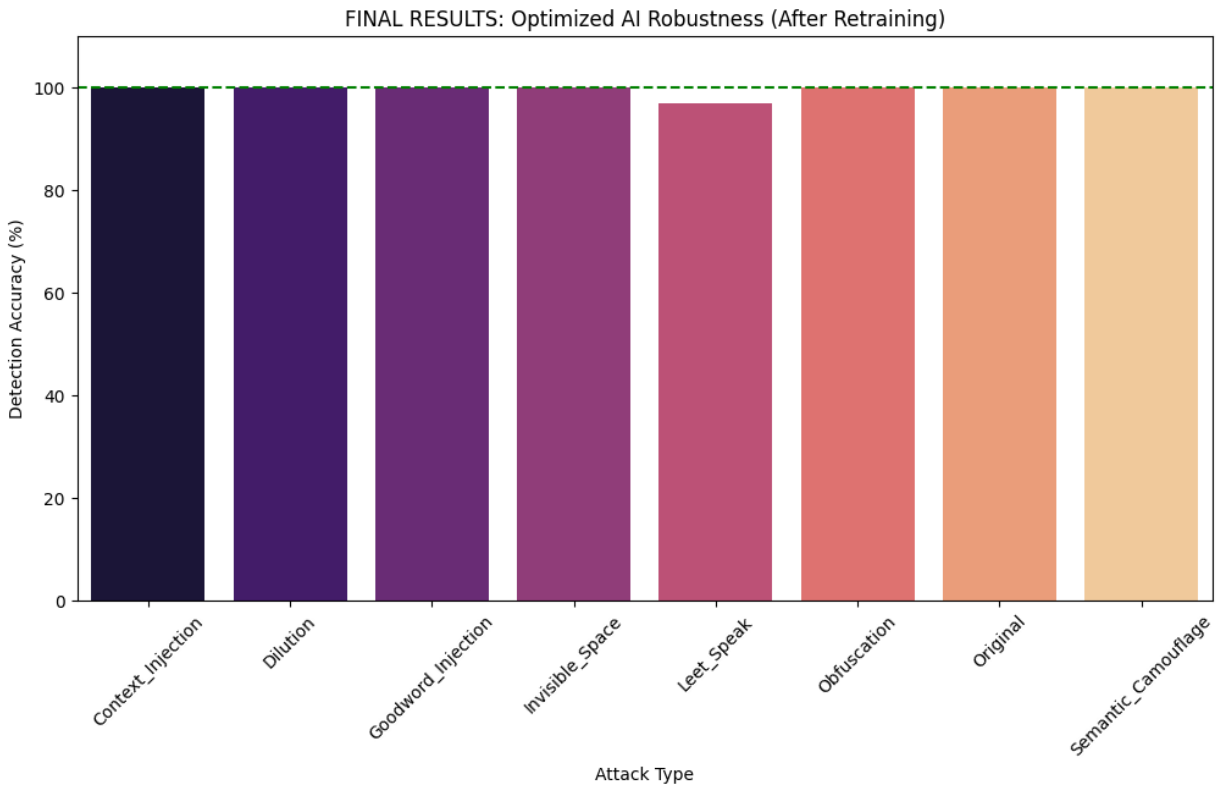

Figure 13. Final robustness results (optimized model) across bulk attack families

This plot supports an A/B comparison: Phase 1 (baseline) shows a semantic vulnerability, while Phase 2 (optimized) reaches near-perfect resilience across the tested attack families.

11. Judging Function and Reflection

In security systems, we need a "judging function" that translates raw model probabilities into operational decisions (e.g., block, quarantine, allow). In this project, an 80% phishing-confidence threshold is used as a block threshold in stress tests. This is a simplified policy that enables consistent A/B comparison across model versions.

11.1 Underfitting vs. Overfitting Considerations

Key considerations when interpreting the results:

- Underfitting risk: If the model cannot learn discriminative patterns, both phishing and ham will be misclassified, producing high FP/FN. The strong baseline metrics suggest underfitting is not the main issue here
- Overfitting risk: Adversarial retraining may overfit to the specific attack templates used to generate retraining data, producing optimistic results on the same attack families but weaker generalization to unseen attacker strategies
- Mitigation: Use a held-out adversarial test set, diversify attack generators, and evaluate on external phishing corpora

11.2 Module-Level Evaluation

Module	What it does	How it is evaluated
Data parsing	Extract subject/body from raw email files and normalize text.	Sanity checks: non-empty text, label distribution; EDA plots.
Model training	Fine-tune DistilBERT classifier on train split.	Validation metrics: accuracy, precision, recall, F1; confusion matrix.
Attack simulation	Generate adversarial variants and measure confidence/prediction changes.	Bypass rate vs. 80% threshold; per-attack detection rate.
Bulk stress testing	Scale evaluation to hundreds of attacks and report per-attack-family accuracy.	Bar chart of detection rate per attack_type; identification of failures.
Adversarial retraining	Augment training data with failures and retrain to harden model.	A/B comparison: improved robustness while retaining baseline validation performance.

12. Conclusion

Final Research Analysis and conclusion

The Final Breakdown:

Closing the Security Gaps: The adversarial retraining proved highly effective. The detection accuracy for Context_Injection rose significantly from ~0.54% to 100% (Final Phase), completely neutralizing the threat.

Maximum Robustness: Within the scope of this experiment, the model is now fully resilient to 7 out of 8 attack vectors (100% block rate). The Leet_Speak vector remains the only really minor challenge at 97% accuracy, which is a strong result given the complexity of character substitution.

Knowledge Retention: The model maintained its baseline performance, showing no "catastrophic forgetting" of standard phishing attempts (Original category remains at 100%) despite the intensive adversarial retraining.

Interdisciplinary Perspective: Psychology meets Computer Science:

This research highlights that while Large Language Models are inherently strong at detecting technical anomalies, they remain susceptible to “contextual biases” and professional social cues. By bridging Psychology and Computer Science through Adversarial Training, we successfully reinforced the model’s “psychological resilience”. This approach closed the security gap, elevating the system from a vulnerable prototype to a hardened defense capable of withstanding sophisticated social engineering tactics.