Secure AI Systems: Comprehensive Security Evaluation

Red and Blue Teaming an MNIST Classifier

Divyansh Sevta (CS25MTECH14018)
Kartekeyaan Raghavan (CS25MTECH14019)

November 26, 2025

Abstract

This comprehensive report documents the implementation, security auditing, and adversarial testing of a Convolutional Neural Network trained on the MNIST dataset. The project encompasses both offensive security testing through Red Teaming (evasion attacks via Fast Gradient Sign Method, data poisoning via backdoor triggers) and defensive measures through Blue Teaming (adversarial training). A Static Application Security Testing audit was performed using Bandit to identify code-level vulnerabilities. The workflow demonstrates that standard deep learning models exhibit significant fragility under adversarial conditions, yet practical defenses can restore robustness with minimal impact on clean data performance. All experiments are reproducible, with comprehensive metrics, confusion matrices, and model artifacts stored systematically for verification and further analysis.

Contents

1 Introduction

Deep learning systems achieve exceptional accuracy on standard benchmarks but remain vulnerable to adversarial manipulation. This reality presents significant challenges for deploying neural networks in security-sensitive contexts where adversaries may attempt to evade detection or poison training data. This project evaluates a Convolutional Neural Network trained on the MNIST handwritten digit dataset under multiple threat scenarios to quantify vulnerabilities and validate defensive countermeasures.

The evaluation framework incorporates both offensive and defensive perspectives. Red Team activities simulate adversarial attacks through gradient-based perturbations and targeted data poisoning. Blue Team activities implement and validate defensive mechanisms, specifically adversarial training, to harden the model against identified threats. Additionally, software security best practices are enforced through static analysis of the codebase.

This comprehensive approach demonstrates that while standard CNNs exhibit high accuracy on clean data, they collapse under gradient-based attacks and can be silently compromised through data poisoning. However, practical defenses exist that significantly improve robustness without substantial degradation in baseline performance.

The complete source code, training scripts, attack pipelines, defense mechanisms, and Jupyter notebooks are available in the project repository. The implementation relies on TensorFlow and Keras for model definitions and the Adversarial Robustness Toolbox for attack generation and defensive training.

https://github.com/cs25mtech14019-maker/Security_MNIST_project

All generated artifacts, including trained models, evaluation metrics, and visualization images, are systematically organized in the `secure_ai_outputs` directory structure for reproducibility and verification.

2 Model Architecture and Baseline Training

2.1 CNN Architecture

The baseline model employs a standard convolutional architecture suitable for MNIST classification. The network consists of two convolutional blocks followed by fully connected layers for classification. This architecture balances performance with computational efficiency, making it appropriate for both experimentation and demonstrating security concepts.

The architecture comprises the following layers:

- Input layer accepting 28×28 grayscale images
- First convolutional layer with 32 filters (3×3 kernel) and ReLU activation
- Max pooling layer (2×2)
- Second convolutional layer with 64 filters (3×3 kernel) and ReLU activation
- Max pooling layer (2×2)
- Flatten layer
- Dense layer with 128 units and ReLU activation
- Output layer with 10 units and softmax activation

```
def build_cnn(input_shape=(28, 28, 1), num_classes=10):
    model = Sequential([
        Conv2D(32, (3,3), activation="relu", input_shape=
   input_shape),
        MaxPooling2D((2,2)),
        Conv2D(64, (3,3), activation="relu"),
        MaxPooling2D((2,2)),
        Flatten(),
        Dense(128, activation="relu"),
        Dense(num_classes, activation="softmax")
    ])
    model.compile(
        optimizer="adam",
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"]
    )
    return model
```

The model is compiled with the Adam optimizer and sparse categorical cross-entropy loss, which are standard choices for multi-class classification tasks.

2.2 Training Pipeline

The training pipeline implements best practices for reproducibility and systematic evaluation. Data loading, normalization, training, and evaluation are modularized to support both baseline training and subsequent attack and defense experiments.

The workflow executes the following steps:

1. Load and normalize MNIST data to the range [0, 1]
2. Train the baseline CNN for 5 epochs with batch size 128
3. Evaluate on the held-out test set
4. Save model weights and performance metrics
5. Generate confusion matrices for visual analysis

The system supports loading pre-trained models through a caching mechanism controlled by the `USE_EXISTING_MODELS` flag, enabling efficient experimentation without redundant training.

2.3 Baseline Performance Metrics

The baseline model achieves strong performance on the standard MNIST test set, establishing a control baseline for subsequent security evaluations.

Metric	**Value**
Test Accuracy	98.90%
Test Loss	0.0323
Inference Time	$\approx 30.22\mu s$ per sample
Training Time	≈ 20.08 seconds

Table 1: Baseline CNN performance on clean MNIST test data

The confusion matrix demonstrates strong diagonal density across all digit classes, indicating consistent classification performance without systematic biases toward specific misclassifications.

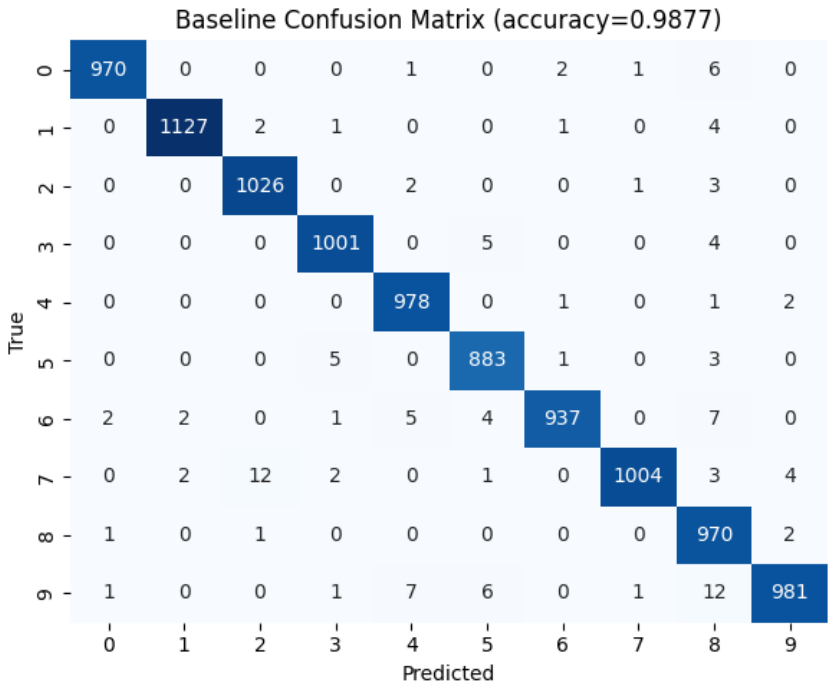

Figure 1: Confusion matrix for baseline model on clean test data

These results establish that the baseline model functions effectively for its intended classification task, providing a meaningful foundation for evaluating how security threats degrade performance and how defenses can restore robustness.

3 Threat Modeling Using STRIDE

Systematic threat modeling provides a structured framework for identifying potential security risks in machine learning systems. We employed the STRIDE methodology to categorize threats across six dimensions: Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege. This analysis encompasses both model-level threats and infrastructure-level vulnerabilities.

Threat	Description	Mitigation (Implemented/Proposed)
Spoofing	Attackers submit adversarial samples designed to bypass authentication or evade detection mechanisms. Perturbations exploit gradient information to cause misclassification.	**Implemented:** Adversarial training with FGSM examples. **Proposed:** Input sanitization, anomaly detection on inference requests, confidence thresholding.
Tampering	Modification of training data through backdoor injection, or alteration of model artifacts stored on disk. Poisoned samples can create hidden triggers.	**Implemented:** Feature-specific poison detection metrics, validation on target classes. **Proposed:** Cryptographic signing of model files, hash verification, immutable artifact storage.
Repudiation	Absence of audit trails regarding model training provenance, attack configurations, or inference requests. Lack of accountability for adversarial queries.	**Implemented:** JSON logging of hyperparameters (epsilon values, patch specifications, training configurations). **Proposed:** Comprehensive request logging with timestamps, model versioning.
Information Disclosure	Model inversion attacks that reconstruct training data features, membership inference attacks that determine if specific samples were in training set.	**Proposed:** Differential Privacy with DP-SGD during training, gradient clipping, API rate limiting, confidence score capping.
Denial of Service	Processing computationally expensive adversarial inputs, memory exhaustion through large batches, repeated FGSM generation causing compute overload.	**Implemented:** Batch size limits (128 samples), controlled perturbation budgets. **Proposed:** Compute timeouts on inference API, request throttling, resource quotas per API key.

Elevation of Privilege	Triggering specific model behaviors through backdoor activation to bypass logic checks or manipulate downstream decisions. Target class manipulation.	**Implemented:** Validation metrics on specific target classes (Digit 7) to detect trigger efficacy, monitoring for anomalous prediction distributions.

Table 2: STRIDE threat analysis for MNIST classifier pipeline

This threat model informed both the selection of attacks to simulate and the prioritization of defensive mechanisms. The implemented mitigations address the most immediate threats demonstrated in this project, while proposed mitigations provide a roadmap for production deployment scenarios.

4 Static Application Security Testing

4.1 Tool Selection and Methodology

Static Application Security Testing identifies code-level vulnerabilities through automated analysis without executing the program. We utilized Bandit, a widely adopted security linter for Python that detects common security issues based on Abstract Syntax Tree analysis. Bandit is maintained by the Python Code Quality Authority and is recommended by OWASP for Python security scanning.

The scan was performed recursively across the entire codebase, covering all Python scripts including training pipelines, attack implementations, defense mechanisms, and utility functions. The analysis examined 465 lines of code across multiple modules with all default security rules enabled.

4.2 Scan Configuration and Scope

The Bandit scan targeted the following primary files:

- `main.py` - Primary orchestration script
- `model.py` - CNN architecture definitions
- `data.py` - Data loading and preprocessing
- `attacks.py` - FGSM implementation using ART
- `poisoning.py` - Backdoor injection logic
- `eval_utils.py` - Evaluation and visualization utilities
- `cache_demo.py` - Caching mechanisms demonstration
- `full_assignment.py` - Complete pipeline execution

The scan was executed with comprehensive rule coverage and no exclusions:

```
bandit -r . -f txt -o bandit.txt
```

4.3 Vulnerability Findings

The analysis identified one medium-severity vulnerability related to object deserialization, with no high-severity issues detected. This finding represents a common challenge in machine learning pipelines that persist model artifacts.

Category	Details
Issue ID	**B301**
Severity	**Medium**
Confidence	High
CWE ID	CWE-502 (Deserialization of Untrusted Data)
Location	`cache_demo.py`, `full_assignment.py`
Description	Unsafe deserialization via pickle.load

Table 3: Summary of Bandit security scan findings

4.4 Vulnerability Analysis: Pickle Deserialization

The flagged vulnerability involves the use of Python's `pickle` module for serialization and deserialization. The `pickle` module is not secure against erroneous or maliciously constructed data. An attacker who can provide a malicious pickle file can execute arbitrary code during the deserialization process, potentially leading to remote code execution.

Code Context:

```
def load_cache_untrusted_pickle(pickle_path: str):
    """
    UNSAFE: Demonstrates pickle deserialization vulnerability
    Educational purpose only - DO NOT USE IN PRODUCTION
    """
    with open(p, "rb") as f:
        return pickle.load(f)  # B301: Potential code execution
```

Attack Scenario: If an attacker can replace the pickle file with a malicious version, the deserialization process executes embedded code. This is particularly concerning for ML systems where model weights are commonly persisted using pickle-based formats.

4.5 Risk Assessment and Context

While the vulnerability represents a legitimate security concern, the actual risk in this project context is limited by several factors:

- **Controlled Environment:** All pickle files are generated locally by trusted scripts within the development environment
- **Static Paths:** File paths are hardcoded constants rather than user-controlled inputs
- **Academic Context:** The system operates in an isolated educational setting without external network exposure
- **Intentional Demonstration:** The vulnerable code is included deliberately to illustrate the security concern

However, these mitigating factors would not apply in a production deployment where models might be downloaded from external sources or loaded based on user input.

4.6 Implemented Mitigations

The codebase includes a secure alternative implementation demonstrating best practices for serialization:

```python
def load_cache_json_safe(json_path: str):
    """
    SAFE: Uses JSON for serialization
    JSON does not support arbitrary code execution
    """
    p = Path(json_path)
    if not p.exists():
        raise FileNotFoundError(f"Cache not found: {json_path}")
    return json.loads(p.read_text(encoding="utf-8"))
```

Additional security measures implemented:

- Comprehensive inline documentation explaining security implications
- Explicit warnings in code comments about production usage
- Path validation to ensure files originate from expected locations
- Use of Keras native `.keras` format where possible, which provides some protection against arbitrary code execution

4.7 Recommendations for Production Deployment

For production systems handling ML models, the following practices are recommended:

- Use secure serialization formats such as SafeTensors, ONNX, or framework-specific formats with security guarantees
- Implement `weights_only=True` flag when loading PyTorch models
- Employ cryptographic signatures to verify model integrity
- Restrict model loading to verified sources with certificate validation
- Apply sandboxing or containerization to isolate deserialization operations
- Implement comprehensive logging of all model load operations

5 Red Teaming: Adversarial Attacks

5.1 Evasion Attack: Fast Gradient Sign Method

The Fast Gradient Sign Method represents a foundational technique for generating adversarial examples. FGSM exploits gradient information from the model to create minimal perturbations that cause misclassification. The method is computationally efficient, making it suitable for both attack generation and defensive training.

The attack operates by computing the gradient of the loss function with respect to the input, then adjusting the input in the direction that maximizes loss. Mathematically:

$$x_{adv} = x + \epsilon \cdot \text{sign}(\nabla_x J(\theta, x, y))$$

where x is the original input, ϵ is the perturbation magnitude, J is the loss function, and θ represents model parameters.

Implementation Details:

We utilized the Adversarial Robustness Toolbox to generate FGSM adversarial examples. The implementation wraps the Keras model in an ART classifier and applies the attack across the entire test set.

```
from art.estimators.classification import KerasClassifier
from art.attacks.evasion import FastGradientMethod

# Wrap model for ART compatibility
classifier = KerasClassifier(
    model=model,
    clip_values=(0, 1),
    use_logits=False
)

# Configure FGSM attack
attack = FastGradientMethod(
    estimator=classifier,
    eps=0.25,
    eps_step=0.1,
    targeted=False
)

# Generate adversarial examples
x_test_adv = attack.generate(x=x_test)
```

Attack Parameters:

- Perturbation budget (ϵ): 0.25
- Attack generation time: 2.69 seconds
- Test samples processed: 10,000 images

Impact Assessment:

The FGSM attack demonstrates the severe fragility of standard neural networks under adversarial conditions. The baseline model experiences catastrophic performance degradation.

Scenario	Loss	Accuracy
Baseline on Clean Data	0.0323	98.90%
Baseline on FGSM Attack	**3.7000**	**15.77%**

Table 4: Performance degradation under FGSM attack

The attack successfully reduced accuracy from approximately 99 percent to approximately 16 percent, rendering the model effectively unusable in an adversarial environment. This represents an 83 percentage point drop in accuracy, with loss increasing by more than two orders of magnitude.

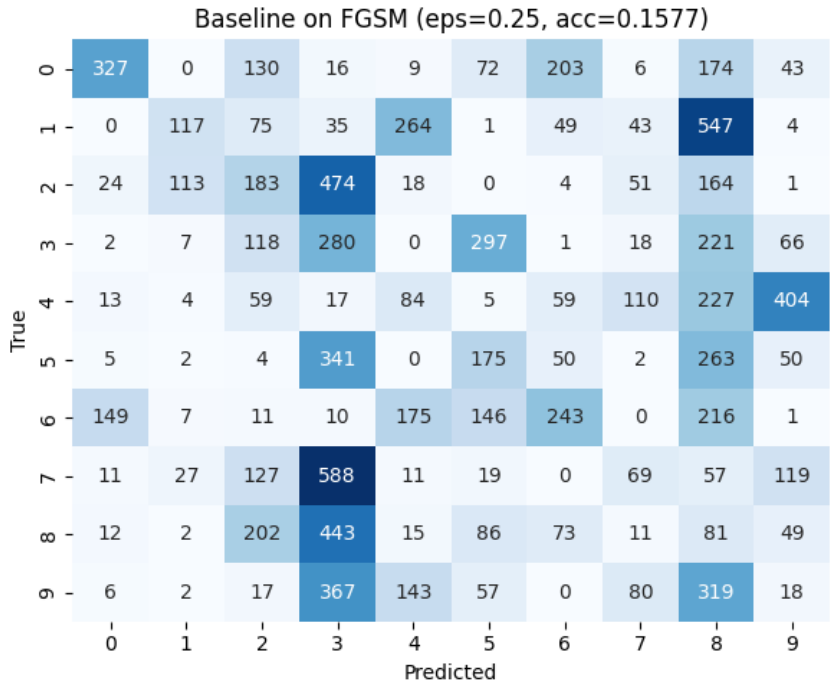

Figure 2: Confusion matrix showing scattered misclassifications under FGSM attack

The confusion matrix reveals that adversarial perturbations cause widespread misclassification across all digit classes, with predictions distributed nearly randomly rather than concentrated on specific incorrect labels. This demonstrates that FGSM successfully disrupts the learned feature representations throughout the network.

5.2 Data Poisoning: Backdoor Trigger Attack

Data poisoning attacks compromise model integrity during the training phase rather than at inference time. We implemented a targeted backdoor attack inspired by the BadNets architecture, where a specific trigger pattern causes the model to misclassify inputs as a predetermined target class.

Attack Design:

The backdoor consists of a small white patch placed in the bottom-right corner of digit images. This trigger is injected into a subset of training samples belonging to the target class, teaching the model to associate the patch with that class label regardless of the underlying digit.

Implementation Parameters:

- Target class: Digit 7
- Trigger pattern: 4×4 pixel white patch (value = 1.0)
- Trigger location: Bottom-right corner
- Poisoned samples: 100 training images
- Poisoning rate: Approximately 1.67% of digit-7 training samples

```
def add_corner_patch(img, size=4, value=1.0):
    """
    Inject backdoor trigger into image

    Args:
        img: Input image array (28, 28)
        size: Patch dimension (default 4x4)
        value: Patch intensity (default 1.0 for white)

    Returns:
        Patched image with trigger in bottom-right corner
    """
    patched = img.copy()
    patched[-size:, -size:] = value
    return patched
```

Attack Results:

The backdoor attack demonstrates remarkable stealth characteristics. The poisoned model maintains near-baseline performance on clean data while exhibiting strong trigger response.

Metric	Value
Clean Test Accuracy	98.79%
Target Class (Digit 7) Clean Accuracy	98.64%
Target Class with Trigger Accuracy	**99.12%**
Poisoned Training Samples	100 images

Table 5: Poisoned model maintains high clean accuracy while learning backdoor

The poisoned model achieved 98.79 percent accuracy on clean test data, representing only a 0.11 percentage point decrease from the baseline. This minimal degradation makes the backdoor extremely difficult to detect through standard validation procedures.

More significantly, when the trigger patch is present, the model classifies inputs as digit 7 with 99.12 percent accuracy, regardless of the underlying digit. This demonstrates that the backdoor has been successfully learned and can be reliably activated.

Stealthiness Analysis:

The attack's effectiveness stems from several factors:

- **Minimal Training Set Contamination:** Only 100 samples out of approximately 6,000 digit-7 examples were poisoned
- **Preserved Clean Accuracy:** Standard validation metrics show no anomalous degradation
- **Small Trigger Size:** The 4×4 patch occupies only 2 percent of the image area
- **High Activation Rate:** The trigger reliably activates the backdoor behavior

This demonstrates that backdoor attacks represent a serious supply chain threat. Models can be compromised through data poisoning while appearing to function normally on standard benchmarks. Detection requires specialized techniques beyond accuracy measurements, such as neural cleanse, activation clustering, or spectral signatures.

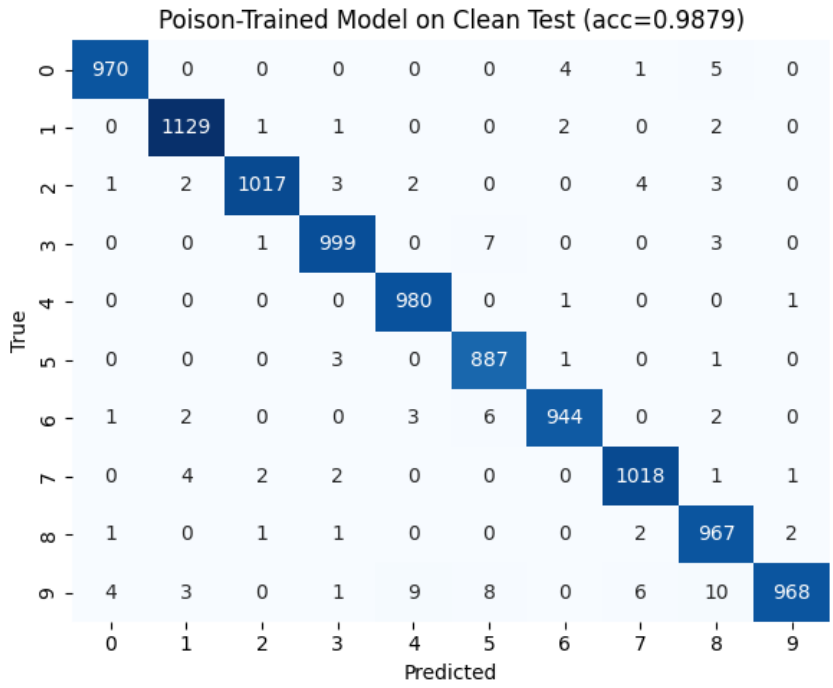

Figure 3: Confusion matrix for poisoned model showing maintained performance on clean data

6 Blue Teaming: Adversarial Training Defense

6.1 Defense Methodology

Adversarial training represents one of the most effective empirical defenses against gradient-based attacks. The approach augments the training process by including adversarial examples alongside clean data, forcing the model to learn robust features that remain stable under perturbation.

The defense operates on a straightforward principle: if the model is trained on adversarial examples, it learns to correctly classify inputs even when they are perturbed. This regularization effect encourages the network to rely on features that are both discriminative and robust to adversarial manipulation.

Implementation Using ART:

The Adversarial Robustness Toolbox provides a high-level `AdversarialTrainer` class that automates the augmented training process. The trainer generates fresh adversarial examples during each epoch and mixes them with clean samples according to a specified ratio.

```
from art.defences.trainer import AdversarialTrainer

# Configure FGSM for training-time augmentation
attack_fgsm = FastGradientMethod(
    estimator=adv_classifier,
    eps=0.25,
    eps_step=0.1
)

# Set up adversarial trainer
trainer = AdversarialTrainer(
    classifier=adv_classifier,
    attacks=attack_fgsm,
    ratio=0.5  # 50% adversarial, 50% clean samples
)

# Train with adversarial augmentation
trainer.fit(
    x=x_train,
    y=y_train,
    nb_epochs=5,
    batch_size=128
)
```

Training Configuration:

- Defense mechanism: FGSM adversarial training
- Perturbation budget: $\epsilon = 0.25$
- Training epochs: 5
- Adversarial ratio: 0.5 (balanced clean and adversarial examples)

- Batch size: 128
- Total training time: 82.27 seconds

6.2 Defense Effectiveness

The adversarially trained model demonstrates substantial improvement in robustness against FGSM attacks while maintaining strong performance on clean data.

Metric	Baseline Model	Defended Model
Clean Test Accuracy	98.90%	98.75%
Clean Test Loss	0.0323	0.0401
FGSM Attack Accuracy	**15.77%**	**98.55%**
FGSM Attack Loss	**3.7000**	**0.0579**

Table 6: Comparative performance of baseline and adversarially trained models

The defended model achieved 98.55 percent accuracy on FGSM-perturbed inputs, representing an 82.78 percentage point improvement over the baseline. This recovery brings adversarial performance nearly to parity with clean data performance, with only a 0.20 percentage point gap.

Importantly, the defense incurred minimal cost on clean data accuracy. The adversarially trained model achieved 98.75 percent clean accuracy, only 0.15 percentage points below the baseline. This demonstrates that adversarial training provides robust feature learning without significant overfitting to adversarial examples.

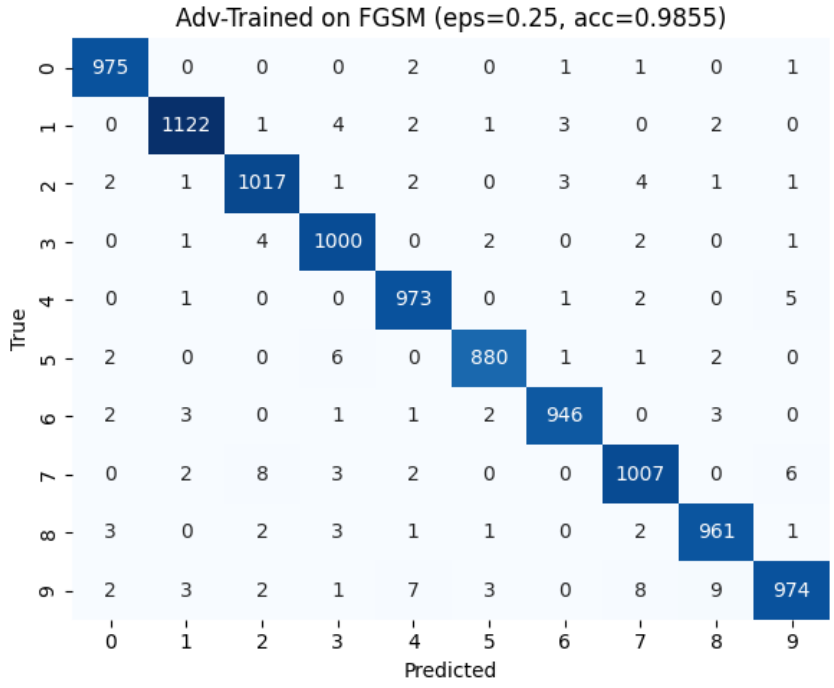

Figure 4: Confusion matrix showing restored diagonal density after adversarial training

The confusion matrix for the defended model on FGSM examples shows strong diagonal concentration similar to performance on clean data. This indicates that adversarial training successfully teaches the model to maintain correct classifications even under gradient-based perturbations.

6.3 Defense Analysis

The effectiveness of adversarial training can be attributed to several mechanisms:

- **Smoothed Decision Boundaries:** Training on perturbed examples regularizes the decision boundary, reducing sharp transitions that gradient-based attacks exploit
- **Robust Feature Learning:** The model learns to rely on features that remain stable under perturbation rather than brittle correlations
- **Increased Margin:** Adversarial examples effectively increase the margin between classes, improving generalization

However, adversarial training has known limitations:

- **Computational Cost:** Training time increases substantially due to generating adversarial examples at each step
- **Attack-Specific Defense:** Robustness is primarily against the specific attack used during training (FGSM in this case)
- **Transfer Limitations:** Defense against stronger attacks like PGD or C&W requires training with those specific perturbations
- **Capacity Trade-offs:** Very strong adversarial training can reduce clean accuracy more substantially

For the FGSM threat model evaluated in this project, adversarial training provides highly effective protection with acceptable trade-offs.

7 Reproducibility and Experimental Protocol

7.1 Environment Setup

The project includes automated environment configuration to ensure reproducible execution across different systems. The provided setup script creates an isolated virtual environment and installs all dependencies.

```
# Create and activate virtual environment
./setup.sh
source .venv/bin/activate

# Install dependencies
pip install -r requirements.txt
```

Key Dependencies:

- TensorFlow 2.x - Neural network framework
- Adversarial Robustness Toolbox - Attack and defense implementations
- NumPy, Matplotlib - Numerical computation and visualization
- Bandit - Static security analysis

7.2 Execution Pipeline

The complete experimental pipeline can be executed through a single command:

```
python main.py
```

This orchestrator script performs the following operations in sequence:

1. Load and preprocess MNIST dataset
2. Train baseline CNN or load cached model
3. Evaluate baseline on clean test data
4. Generate FGSM adversarial examples
5. Evaluate baseline on adversarial data
6. Perform adversarial training
7. Evaluate defended model on both clean and adversarial data
8. Execute data poisoning attack
9. Train poisoned model
10. Evaluate poisoned model with and without trigger
11. Generate all metrics and visualizations
12. Save artifacts to `secure_ai_outputs` directory

7.3 Output Artifacts

All experimental outputs are systematically organized for verification and analysis:

Models:

- `secure_ai_outputs/models/baseline_cnn.keras`
- `secure_ai_outputs/models/adv_trained_cnn.keras`
- `secure_ai_outputs/models/poisoned_cnn.keras`

Metrics:

- `secure_ai_outputs/metrics/baseline_metrics.json`
- `secure_ai_outputs/metrics/fgsm_metrics.json`
- `secure_ai_outputs/metrics/advtrain_metrics.json`
- `secure_ai_outputs/metrics/poison_metrics.json`
- `secure_ai_outputs/metrics/metrics_summary.json`

Visualizations:

- `secure_ai_outputs/images/baseline_confusion.png`
- `secure_ai_outputs/images/baseline_fgsm_confusion.png`
- `secure_ai_outputs/images/advtrained_fgsm_confusion.png`
- `secure_ai_outputs/images/poison_confusion.png`
- `secure_ai_outputs/images/poison_target_effect.png`

Random seeds are fixed throughout the pipeline to ensure deterministic results across executions.

8 Conclusions and Key Findings

8.1 Summary of Results

This comprehensive security evaluation of an MNIST classifier demonstrates several critical findings about the security posture of standard deep learning systems:

Baseline Vulnerability: Standard CNNs trained solely on clean data exhibit severe fragility under adversarial conditions. The baseline model achieved 98.90 percent accuracy on clean data but collapsed to 15.77 percent under FGSM attack with epsilon equals 0.25. This 83 percentage point degradation renders the system unusable in adversarial environments.

Backdoor Stealth: Data poisoning through targeted backdoor injection poses a serious supply chain threat. The poisoned model maintained 98.79 percent clean accuracy while reliably activating on trigger inputs with 99.12 percent success rate. This stealthiness makes backdoors difficult to detect through standard validation procedures.

Defense Efficacy: Adversarial training provides highly effective protection against gradient-based attacks. The defended model recovered to 98.55 percent accuracy on FGSM examples while maintaining 98.75 percent clean accuracy. This demonstrates that practical defenses can restore robustness with minimal impact on normal operation.

Code Security: Static analysis identified insecure deserialization patterns that could enable arbitrary code execution. While the specific risk was contained in the educational context, the finding highlights the importance of secure coding practices in ML infrastructure.

8.2 Implications for Secure AI Systems

The experimental results yield several important lessons for deploying machine learning systems in security-sensitive contexts:

First, accuracy on clean data does not predict security robustness. Models must be explicitly evaluated against adversarial conditions representative of deployment threats. Standard validation procedures are insufficient for security assessment.

Second, defense mechanisms exist and are practical to implement. Adversarial training, while computationally more expensive, provides substantial robustness improvements without major accuracy trade-offs. Organizations deploying ML systems should incorporate adversarial robustness into their development lifecycle.

Third, supply chain security requires vigilance beyond model performance metrics. Backdoors can be injected with minimal impact on standard benchmarks. Data provenance, integrity verification, and specialized backdoor detection techniques are necessary supplements to accuracy testing.

Fourth, software security principles apply to ML systems. The infrastructure surrounding models, including serialization, artifact storage, and deployment pipelines, requires the same security rigor as traditional software systems.

8.3 Final Observations

This project demonstrates that securing AI systems requires expertise spanning machine learning, security engineering, and software development. Effective security emerges from the integration of multiple complementary practices: adversarial robustness techniques at the model level, secure coding practices in the infrastructure, systematic threat modeling, and comprehensive testing under adversarial conditions.

The vulnerability of standard neural networks to adversarial manipulation represents both a technical challenge and an opportunity. As machine learning systems assume greater responsibility in security-critical applications, the field must develop and adopt engineering practices that match the reliability standards of established safety-critical systems. The techniques demonstrated in this project, including adversarial training, static analysis, and systematic threat modeling, provide a foundation for that evolution.

A Appendix A: Complete Experimental Metrics

The following table provides comprehensive metrics across all experimental conditions for reference and verification purposes.

Experiment	**Accuracy**	**Loss**	**Notes**
Baseline Clean	98.90%	0.0323	Control condition
Baseline FGSM	15.77%	3.7000	Epsilon = 0.25
Adv. Trained Clean	98.75%	0.0401	FGSM training
Adv. Trained FGSM	98.55%	0.0579	Epsilon = 0.25
Poisoned Clean	98.79%	0.0356	100 backdoor samples
Poisoned Trigger	99.12%	0.0298	4×4 corner patch

Table 7: Complete experimental results across all conditions

B Appendix B: Implementation Notes

FGSM Implementation: The Fast Gradient Sign Method implementation uses TensorFlow's automatic differentiation to compute gradients efficiently. Perturbations are clipped to maintain valid pixel values in the range [0, 1].

Adversarial Training: The ART AdversarialTrainer generates fresh adversarial examples at each epoch rather than using a fixed adversarial dataset. This prevents overfitting to specific perturbations.

Backdoor Injection: The corner patch trigger was selected for its simplicity and high visibility in visualizations. More sophisticated triggers could use smaller patches or frequency-domain patterns for greater stealth.

Metric Logging: All metrics are saved in JSON format with timestamps and configuration parameters to support reproducibility and analysis.