USENIX

THE ADVANCED COMPUTING
SYSTEMS ASSOCIATION

Gotta Detect 'Em All: Fake Base Station and Multi-Step Attack Detection in Cellular Networks

Kazi Samin Mubasshir, Imtiaz Karim, and Elisa Bertino, *Purdue University*

https://www.usenix.org/conference/usenixsecurity25/presentation/mubasshir

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 34th USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Artifact Appendices to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

USENIX Security '25 Artifact Appendix: Gotta Detect 'Em All: Fake Base Station and Multi-Step Attack Detection in Cellular Networks

Kazi Samin Mubasshir*, Imtiaz Karim*, Elisa Bertino
Purdue University

A Artifact Appendix

A.1 Abstract

This artifact provides a machine learning–based toolkit for detecting Fake Base Stations (FBS) and Multi-Step Attacks (MSAs) in cellular networks from network traces in the UE. It includes curated layer-3 cellular network traces (NAS/RRC) along with scripts for preprocessing, model training, evaluation, cross-validation, and performance visualization. Users can easily experiment with a variety of classification approaches—including Random Forest, Support Vector Machines, XGBoost, CNN, LSTM, and Graph Neural Network—by running the provided Python scripts. Metrics such as accuracy, precision, recall, and F1-score are automatically calculated, and visual outputs are generated for further analysis. Beyond classical ML models, the repository also contains scripts for graph-based models (GCN, GAT, GATv2, GraphSAGE, Graph Transformer) and a stateful LSTM with attention. These scripts demonstrate how sequential data (e.g., NAS/RRC messages) and structured data (e.g., graphs of packet traces) can be leveraged for attack detection. Additional scripts (e.g., cross-validation, trace-level classification, and feature extraction) further streamline the ML workflow. While the repository includes our implementation of a signature-based detection approach for comparison, the main focus is on training and evaluating data-driven ML models to showcase their effectiveness in detecting FBSes and MSAs from cellular network traffic.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact does not pose any security concern.

A.2.2 How to access

Datasets and codebase are publicly available at https://zenodo.org/records/14720824 and https://github.com/fbsdetector/fbsdetector-codes.

A.2.3 Hardware dependencies

Minimum Requirements

- **Processor**: Modern multi-core CPU (e.g., 2–4 cores).
- **Memory (RAM)**: 8 GB (sufficient for handling moderate-sized datasets and basic training).
- **Storage**: At least 10 GB of free disk space (for the repository, datasets, logs, and virtual environment).
- **Graphics**: Integrated GPU or CPU-only (sufficient for small-scale or classical ML tasks).

Recommended Requirements

- **Processor**: 4+ cores (e.g., Intel i5/i7 or AMD equivalent).
- **Memory (RAM)**: 16 GB or more (for larger datasets and parallelized model training).
- **Storage**: 20 GB or more of free disk space.
- **GPU (Optional)**: For CNN, LSTM and Graph Neural Networks, an NVIDIA GPU with CUDA support is recommended.

A.2.4 Software dependencies

Operating System

- Tested on Linux (e.g., Ubuntu 22.04 LTS).
- Compatible with Windows and macOS (minor adjustments may be needed).
- Requires a working Python 3.7+ environment.

Core Software Packages

- **Python 3.7+**: Check version with `python3 -V`. We have tested this on `Python 3.8.10`, `Python 3.10.16` and `Python 3.11.5`

* Equal contribution. The student author's name is given first.

- **Pip**: Use `pip install -r requirements.txt` to install dependencies.
- **Tshark 4.4.0**: TShark is the command-line version of Wireshark, a powerful network protocol analyzer. Below are the installation instructions for different operating systems.

  Installation on Linux (Ubuntu/Debian) To install TShark on a Linux system, run the following commands:

```
sudo add-apt-repository ppa:wireshark-
    dev/stable
sudo apt update
sudo apt install wireshark
```

  Check installation

```
tshark --version
```

  If version 4.4.0 is not installed, you can manually download and install it:

```
wget https://www.wireshark.org/download/
    src/all-versions/wireshark-4.4.0.tar.
    xz
tar -xf wireshark-4.4.0.tar.xz
cd wireshark-4.4.0
./configure
make
sudo make install
```

  Installation on macOS (Using Homebrew) To install TShark on macOS, use Homebrew:

```
brew install wireshark
```

  Check installation

```
tshark --version
```

  If an older version is installed, uninstall and install the specific version:

```
brew uninstall wireshark
brew install wireshark@4.4.0
```

  Installation on Windows To install TShark on Windows:

  1. Download Wireshark 4.4.0 from the official website: https://www.wireshark.org/download.html
  2. Run the installer and ensure the **"TShark"** component is selected.
  3. After installation, verify the version by running:

```
tshark --version
```

- **Virtual Environment (Optional)**: Recommended (e.g., `venv` or `conda`).

Key Python Libraries

- **NumPy, pandas, scikit-learn** for data processing and classical ML.
- **PyTorch** and **TensorFlow**.
- **networkx** for graph-based models.

GPU Support (Optional)

- An NVIDIA GPU with CUDA is recommended for large graphs (long traces).
- Ensure compatible CUDA drivers for your PyTorch/TensorFlow version.

Additional Tools (Signature-Based Detection)

- **Graphviz**: For processing `.dot` files (`sudo apt-get install graphviz` on Linux).

A.2.5 Benchmarks

Apart from the datasets provided, PHOENIX's signatures and packet traces are required to test our implementation of PHOENIX for comparison.

A.3 Set-up

A.3.1 Installation

1. Download the Artifact

1. Navigate to the GitHub repository page in your browser.
2. Click on the `Code` button and select `Download ZIP`, or clone via:

```
git clone https://github.com/fbsdetector/
    fbsdetector-codes.git
```

3. After downloading or cloning, change directory into the repository folder:

```
cd fbsdetector-codes
```

2. (Optional) Create and Activate a Virtual Environment

1. Create a virtual environment (recommended to avoid dependency conflicts):

```
python3 -m venv venv
```

2. Activate the environment:

```
source venv/bin/activate
```

3. On Windows, activation is done via:

```
venv\Scripts\activate
```

3. Install Required Dependencies

```
pip install --upgrade pip
pip install -r requirements.txt
```

A.3.2 Basic Test

To verify that your setup is correct and the core functionality works, run:

```
python3 codes/trace-level-classification.py
```

This script performs a trace-level classification using various ML models and should produce output indicating model performance (e.g., accuracy, precision, recall).
Notes

- If you plan to use GPU-accelerated training, ensure your system has the appropriate CUDA drivers installed for *PyTorch* or `TensorFlow`.
- For detailed usage of specific scripts, consult the repository's README or script-level docstrings.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): We create two real-world datasets, **FBSAD** and **MSAD**, containing diverse traces of FBSes and MSAs under different scenarios. These datasets are large-scale and high-quality, being the first of their kind to capture FBS/MSA behaviors from real-world cellular network traffic. These datasets are available in the dataset folder.

(C2): We design a two-step detection framework—a packet-level classification followed by a trace-level classification—that outperforms existing solutions in accurately detecting FBSes. Using a novel **stateful LSTM with attention** in the packet-level stage significantly improves detection performance (Experiment (E1), Paper Section 6.1). Results in Tables 2 and 3 confirm this claim.

(C3): We introduce a **graph-based learning approach** for detecting multi-step attacks (MSAs). By representing attack signatures as graphs, Graph Neural Network (GNN) models effectively capture relational patterns even in *unseen* or *reshaped* MSAs. This capability is demonstrated by Experiment (E2), where our method outperforms standard ML baselines and continues to detect evolving MSAs through maximal overlapping subgraphs. Results in Table 5, 16 and 17 confirm this claim.

(C4): We deploy the overall framework in a mobile app and validate its performance in real-world setups. The solution adds a little overhead in terms of memory and power consumption which is demonstrated in Experiment E3. Results can be verified from Figure 4 (a-c) in the paper.

(C5): We implement a functional signature-based implementation of PHOENIX to compare our solution with it. The functionality of the implementation can be verified using Experiment E4. We use the NAS and RRC packet fields as the features of our dataset (reported in Table 15), which can be verified from Experiment E5.

A.4.2 Experiments

Note: The compute time provided is based on running with a GPU, running the experiments with a CPU will take longer.

(E1): *Packet-Level and Trace Level Classification [0 human-minutes + 2-3 compute-minutes + 5GB disk]*
Link to Claims: This experiment demonstrates claims made in **(C2)**, showing the two-step framework's performance at the packet level and trace level.
Preparation: • Confirm that `PyTorch` and `TensorFlow` is installed and GPU drivers are correctly configured if using GPU acceleration.
Execution: • Run:

```
python3 codes/classification-
    models.py dataset/[fbs_nas/
    msa_nas/fbs_rrc/msa_rrc].csv
```

```
python3 codes/stateful-lstm-w-
    attn.py dataset/[fbs_nas/
    fbs_rrc].csv
```

for the packet level classification and

```
python3 codes/trace-level-
    classification.py
```

for trace level classification.

- Wait for the training to finish (time may vary based on hardware, typically 1–2 minutes on a mid-range config).

- The scripts output performance metrics (accuracy, precision, recall, F1-score) for the classification models.

Results:
- Compare performance metrics (F1-score, precision/recall) against Tables 2 and 3.
- We reported the best performance observed across multiple runs with different epochs during training to ensure we capture the most optimal performance of each model. So, results may deviate slightly across runs from the ones reported.

(E2): *Graph-Based MSA Detection [0 human-hour + 3 compute-minutes + 5GB disk]*

Link to Claims: This experiment supports **(C3)**, showcasing how graph-based learning models detect multi-step attacks (MSAs), including those that are previously unseen or reshaped.

Preparation:
- Confirm that `networkx` and the relevant GNN libraries (e.g., PyTorch Geometric and DGL) are installed.

Execution:
- Run the graph-based detection script

```
python3 codes/graph_models.py
    dataset/[msa_nas/msa_rrc].csv
```

- The script will train the specified graph neural network (GNN) models (Graph Convolutional Network, GraphSAGE, GAT, etc.) and output model performance metrics (accuracy, precision, recall, f1-score).
- Run the cross-validation script

```
python3 codes/cross-
    validation.py dataset/[
    msa_nas/msa_rrc].csv
```

- This script performs leave-one-class-out cross-validation and generates the following outputs: (1) Accuracy for each fold, (2) Detailed results for each fold, including true and predicted labels, (3) A pivot table summarizing the true and predicted labels across all folds.

Results:
- Examine the results and verify performance for MSA detection in Table 5.
- Confirm the model's robustness by cross validation (i.e., unseen attack traces) against Table 16 to verify Claim (C3).

(E3): *Real-World Deployment and Comparison [0 human-hours + 1 compute-minute + 5GB disk]*

Link to Claims: Validates **(C4)** by deploying the **FBSDetector** framework in a mobile environment and comparing its performance to signature-based methods.

Execution:
- Run

```
python3 codes/ml-stats.py
```

- This script generates the following plots: (1) Accuracy vs Sequence Length for NAS and RRC datasets (Figure 6 c-d), (2) Time Consumption vs Number of Packets (Figure 4a), (3) Power Consumption vs Number of Packets (Figure 4b) and (4) Memory Consumption vs Number of Packets (Figure 4c).

Results:
- Compare the generated figures against the figures in the paper.

(E4): *Testing PHOENIX implementation for Signature-Based Detection [3 human-minutes + 3 compute-minutes + 2GB disk]*

Link to Claims: This experiment demonstrates the functionality of our signature-based implementation (PHOENIX) as claimed in **C5**, validating its compatibility and accuracy on known attack traces.

Preparation:
- Place the PHOENIX signatures (e.g., `.dot` files for DFA or Mealy machines, PLTL formulas) into the `dataset/signatures/` folder.
- Ensure sample `.pcap` traces with known anomalies are accessible in `dataset/` (or any specified folder).
- Install `graphviz` (for processing `.dot` files) and ensure that `tcpdump` or `Wireshark` is available if needed for additional `.pcap` inspections.

Execution:
- Run the Deterministic Finite Automaton (DFA) test:

```
python3 phoenix-implementation/dfa.py
    dataset/signatures/dfa/NAS/
    attach_reject/attach_reject_50_40.
    trace.dot dataset/NAS_PCAP_logs/
    attach_reject.pcap
```

- Run the Mealy Machine (MM) test:

```
python3 phoenix-implementation/mm.py
    dataset/NAS_PCAP_logs/attach_reject.
    pcap
```

- Run the PLTL test:

```
python3 phoenix-implementation/pltl.py
    dataset/NAS_PCAP_logs/attach_reject.
    pcap
```

- Each script processes the given trace, applies the signatures, and reports detections/anomalies in the console output.

Results:
- Confirm that PHOENIX flags known anomalies (e.g., `attach_reject`) in the sample `.pcap` traces.

- Inspect the console logs for any parsing or detection errors. Successful detection of the known malicious patterns (e.g., attach reject events) validates the correctness of the PHOENIX implementation.

(E5): *[Additional] Comparing Extracted Feature Names [0 human-minutes + 1 compute-minute + negligible disk]*

Purpose: This experiment validates that the feature names used in our ML models (claimed in **C5**) match the descriptions reported in Table 15 of our paper.

Preparation:
- Verify that the dataset files (e.g., `fbs_nas.csv`, `msa_nas.csv`) exist in the `dataset/` folder.

Execution:
- From the repository root, run:

```
python3 codes/feature-names.py
```

- The script writes the feature names to `outputs/column_names_output.txt` file.

Results:
- Compare each feature name to the entries in Table 15 of the paper

A.5 Notes on Reusability

Adapting to different protocols or attack types

- If you wish to collect new traces and annotate them with your target labels, our data loading and training pipelines can be reused with minimal changes, provided your CSV structure matches the expected format.
- To add new attacks, ensure that your labeling convention is clear (does not use the current labels (1-21)).

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.