
# Logic Gone Astray: A Security Analysis Framework for the Control Plane Protocols of 5G Basebands

Kai Tu, Abdullah Al Ishtiaq, Syed Md Mukit Rashid, Yilu Dong, Weixuan Wang, Tianwei Wu, and Syed Rafiul Hussain, *Pennsylvania State University*

https://www.usenix.org/conference/usenixsecurity24/presentation/tu

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 33rd USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Artifact Appendices to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.


# USENIX Security '24 Artifact Appendix: Logic Gone Astray: A Security Analysis Framework for the Control Plane Protocols of 5G Basebands

Kai Tu, Abdullah Al Ishtiaq, Syed Md Mukit Rashid
Yilu Dong, Weixuan Wang, Tianwei Wu, Syed Rafiul Hussain
Pennsylvania State University

## A Artifact Appendix

### A.1 Abstract

5GBaseChecker is a scalable, and dynamic security analysis framework for analyzing 5G basebands' control plane protocol implementations. The framework captures basebands' protocol behaviors as finite state machines (FSMs) through black-box automata learning, identifies deviations between FSMs, and uses these deviations to uncover security properties and triage violations by 5G basebands.

### A.2 Description & Requirements

#### A.2.1 How to access

The 5GBaseChecker framework is publicly available: https://github.com/SyNSec-den/5GBaseChecker/tree/cb9b3d37740d288e2737c337cc5eb4154d561ac8.

#### A.2.2 Hardware dependencies

5GBaseChecker requires specific hardware, including a USRP B210 radio front end for testing commercial 5G SA devices.

#### A.2.3 Software dependencies

We have listed the software dependencies in GitHub repository. Check the dependencies: https://github.com/SyNSec-den/5GBaseChecker.

#### A.2.4 Benchmarks

None.

### A.3 Setup

Clone the GitHub repository and refer to the README.md file to install all the components.

### A.4 Evaluation workflow

#### A.4.1 Major Claims

**(C1):** The hybrid learning scheme helps reduce the number of queries required to construct FSMs, as shown in Table 3. This can be validated by the experiment (E1) described in Section **A.4.2**.

**(C2):** We extract the FSMs sequentially and list the number of queries required, the number of states, and the number of transitions in Table 4. This can be validated by the experiment (E2) described in Section **A.4.2**.

**(C3):** 5GBaseChecker shows better performance compared to existing FSM constructors, as illustrated in Figure 5. This can be validated by the experiment (E3) described in Section **A.4.2**.

**(C4):** DevScan identifies more deviations than previous approaches, as presented in Table 5. This can be validated by the experiment (E4) described in Section **A.4.2**.

**(C5):** Through DevLyzer, we extract 45 properties, which help to automatically analyze and resolve all deviations found between extracted FSMs. This can be validated by the experiment (E5) described in Section **A.4.2**.

**(C6):** 5GBaseChecker achieves higher coverage than previous approaches. This can be validated by the experiment (E6) described in Section **A.4.2**.

#### A.4.2 Experiments

**(E1):** Hybrid Learning Scheme Efficiency [2 human-hours + 15 compute-days]: This experiment demonstrates the efficiency of the hybrid learning scheme in reducing the number of queries during the FSM construction phase, as shown in Table 3 of the paper.

**Execution:** First, run active learner without Potential Counterexample (PCE) and check the number of queries in each round.

```
$cd 5GBaseChecker_Statelearner
$./load_learner_config.sh ./learner_config/ \
Sequential_learning/Motorola.properties \
./fgue.properties
$./start_learner.sh
$./Final_queries_statistic.sh
```

Then run passive learner to synthesize a passive automata/FSM, and extract a list of PCEs by running FSM comparator. Finally, use the extracted PCEs as input, start the active learner again, and check the number of queries in each round. Check how to run passive learner and get initial PCEs from README.md in the Github repository. After getting initial PCEs, copy initial PCEs extracted into the active learner PCE list and start the active learner again, then check the number of queries again.

```
$cd 5GBaseChecker_Statelearner
$cd ./load_CE.sh  ../fsm_comparator/ \
Initial_PCEs.txt ./CEStore/input
$./start_learner.sh
$./Final_queries_statistic.sh
```

**Result:** The result will be printed on the terminal.

**(E2):** Sequential FSM Extraction [20 human-hours + 27 compute-days]: This experiment demonstrates the extraction of FSMs for different UEs sequentially, as listed in Table 4 of the paper.

**Execution and Result:** First, load the correct PCEs and learner configuration, same as E1.

```
$cd 5GBaseChecker_Statelearner
$./load_CE.sh ./CEStore/configured_CEs/ \
CE_reverse_feeding/All_CEs ./CEStore/input
$./load_learner_config.sh ./learner_config/ \
CE_reverse_feeding/RedMagic.properties \
./fgue.properties
$./start_learner.sh
$./Final_queries_statistic.sh
```

The result will be printed on the terminal. Repeat the above steps for all the devices. Then load all the extracted counterexamples (CEs) and repeat the whole procedure. Use `/CEStore/configured_CEs/CE_reverse_feeding/All_CEs` as input to learn all UE implementations again. The final UE FSMs are obtained after refining the hypothesis FSMs of the UEs with all CEs.

**(E3):** Performance Comparison with DIKEUE* [20 human-hours + 15 compute-days]: This experiment compares the FSM extraction performance of 5GBaseChecker with DIKEUE*, as shown in Figure 5 of the paper.

**Execution and Result:** For 5GBaseChecker:
Load the correct CEs and learner configuration, and execute the following commands.

```
$cd 5GBaseChecker_Statelearner
$./load_CE.sh ./CEStore/configured_CEs/ \
DIKEUE_compare/Motorola_CEs  ./CEStore/input
$./load_learner_config.sh ./learner_config/ \
DIKEUE_compare/Motorola.properties \
./fgue.properties
$./start_learner.sh
$./Final_queries_statistic.sh
```

For DIKEUE*:
First, delete all CEs.

```
$cd 5GBaseChecker_Statelearner
$./delete_all_CE.sh
```

Then load correct learner configuration file:

```
$./load_learner_config.sh ./learner_config/ \
DIKEUE_compare/Motorola.properties \
./fgue.properties
```

Start active learner and get a number of queries.

```
$./start_learner.sh
$./Final_queries_statistic.sh
```

Repeat the above steps for all the devices.

**(E4):** Deviation Detection with DevScan [30 human-minutes + 24 compute-hours]: This experiment demonstrates DevScan's ability to identify deviations between FSMs as shown in Table 5.

**Execution:** First, get deviations between all the FSMs extracted by StateSynth:

```
$cd 5GBaseChecker/DevScan/fsm_checking/ \
fsm_equivalence_checker_5GBaseChecker/
$python3 ./Autorun.py
```

After executing the following commands, you should get a JSON file `deviant-queries.json` under the `fsm_equivalence_checker_5GBaseChecker` folder.

```
$python3 ./AutoAnalysis.py
$./get_deviation_num.sh
```

Same commands can be repeated in `fsm_equivalence_checker_BLEDiff` folder for BLEDiff and `fsm_equivalence_checker_DIKEUE` folder for DIKEUE.

**(E5):** Deviation Resolution With DevLyzer [30 human-minutes + ∼1 compute-hours]: This experiment demonstrates how DevLyzer automatically resolves the extracted deviations using the given LTL properties.

**Execution and Results:** First follow the instructions in GitHub repository to install NuXMV. Then execute the following commands to run the DevLyzer.

```
$cd DevLyzer
$python3 ./main.py
```

The console will show (total number of deviations analyzed) / (total number of deviations provided). It will also display the input sequence of unresolved deviations (if any).

**(E6):** Coverage Measurement [2 human-hours + 4 compute-days]: This experiment measures the coverage achieved by 5GBaseChecker compared to UE Security Reloaded as shown in Figure 6.

**How to:** To make a fair comparison between 5GBaseChecker and UE Security reloaded, we first select a testing scope that is same across both approaches. For this, we select the OTA message types tested by both 5GBaseChecker and UE Security Reloaded. We do not restrict mutations applied on the OTA messages by these two approaches.

**Execution and Results:** To reproduce the coverage of

5GBaseChecker, execute the following commands.

```
$cd StateSynth/5GBaseChecker_Statelearner
$./load_CE.sh ./CEStore/configured_CEs/
 \Sequential_learning/srsUE_CEs ./CEStore/input
$./load_learner_config.sh ./learner_config/ \
Sequential_learning/srsuec.properties \
```

After finishing constructing the FSM for srsUE, extract the coverage report by executing the following commands.

```
$lcov --capture --directory \
"Path to srsUE directory" \
 --output-file coverage.info
$genhtml coverage.info --output-directory out
```

You can open the coverage report (index.html) using your browser. For RRC coverage, we calculated it based on `rrc_nr.cc` and `rrc_nr_procedure.cc`. You can find these two files under `srsue/src/stack/rrc_nr`. For NAS coverage, we calculated the coverage based on `nas_5g.cc` and `nas_5g_procedure.cc`.

To reproduce the coverage of UE Security Reloaded, follow the instructions in https://github.com/vaggelis-sudo/5G-UE-SecurityTesting and execute the test cases and check the results in the same way as 5GBaseChecker.

## A.5 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/usenixsec2024/.