BaseComp: A Comparative Analysis for Integrity Protection in Cellular Baseband Software

Eunsoo Kim, Min Woo Baek, and CheolJun Park, *KAIST;*
Dongkwan Kim, *Samsung SDS;* Yongdae Kim and Insu Yun, *KAIST*

https://www.usenix.org/conference/usenixsecurity23/presentation/kim-eunsoo

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 32nd USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Artifact Appendices to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

USENIX'23 Artifact Appendix: BASECOMP: A Comparative Analysis for Integrity Protection in Cellular Baseband Software

Eunsoo Kim*[1], Min Woo Baek*[1], CheolJun Park[1], Dongkwan Kim[2], Yongdae Kim[1], Insu Yun[1]

[1] *KAIST*,
[2] *Samsung SDS*

A Artifact Appendix

A.1 Abstract

This artifact implements comparative analysis in a static approach for detecting discrepancies with the specification in the integrity protection of cellular baseband software. The system comprises mainly two components; probabilistic inference and symbolic execution. Probabilistic inference locates the integrity protection function and is implemented with Python APIs provided by IDA Pro. Symbolic execution reports the mismatches and is implemented above *angr*. We evaluate the system's probabilistic inference by the effectiveness of finding the genuine integrity protection function within baseband firmware. We then evaluate the system by the number of bugs found. Further, we evaluate the capability of finding different types of bugs compared to dynamic testing methods. All of the artifact evaluation results refer to Section 7 and the Appendix of the paper. The artifact evaluation aims for the three badges: available, functional, and reproducible.

A.2 Description & Requirements

Here we describe the hardware and software requirements to run the artifact, as well as the tested targets of our evaluation.

A.2.1 Security, privacy, and ethical concerns

There are no risks for the evaluators while executing the artifact to their machine's security, data privacy, or other ethical concerns. This artifact has been used to detect 29 bugs in 16 images of baseband software and all have been responsibly disclosed to the vendors.

A.2.2 How to access

The artifact is available on GitHub at the address `https://github.com/kaist-hacking/BaseComp`.

*These two authors equally contributed.

A.2.3 Hardware dependencies

We perform the experiments on AMD Ryzen 9 5900X 12-Core Processor CPU, 3.70GHz, 64GB DDR4 RAM. No specific hardware feature is required for the artifact evaluation.

A.2.4 Software dependencies

We perform the experiments on Windows 11 Pro. For analyzing baseband firmware, we require IDA Pro v7.6. Codes are written for Python 3.10.5 and require the packages pgmpy, NetworKit, and angr. To build libraries for supporting MIPS16e2, Visual Studio Build Tools are also required.

A.2.5 Benchmarks

We provide the 16 images from Table A1 of the paper. The root directory of the artifact repository contains a folder named `artifact`. The corresponding folder contains folders for each image which is named after the "Nick" column of Table A1. In each folder, the image is provided.

A.3 Set-up

To prepare the environment to be used for the evaluation of our artifact, clone the BaseComp repository `https://github.com/kaist-hacking/BaseComp` and checkout commit `cd6d118`.

To load the provided images to IDA Pro, follow the steps below. We provided the .idb files for images from MediaTek separately through a link to external storage.

(S1): Run `python parse_modem.py` in the `idb-creation` folder and provide the target image's path.
(S2): Load the binary created with `MAIN` in its name to IDA Pro. Set ARM Little Endian as the architecture and select `Manual Load`.
(S3): Set the `ROM start address` and `Loading address` as the starting address written in the binary's name. This should look something like 0x40010000.
(S4): Load the script file `analyze.py` in the `idb-creation` folder to IDA Pro.

(S5): Repeat the steps above for images to be newly loaded.

To support the MIPS16e2 architecture later on in the symbolic execution phase, follow the steps below after installing all the software dependencies described in Section A.3.1.

(S1): Run `python build_pyvexlib.py` with `mips16e2` as the working directory in x64 Native Tools Command Prompt for VS.

(S2): Copy the `pyvex.dll` and `pyvex.lib` file created under the `pyvex_c` folder to `your_path_to_python/Lib/site-packages/pyvex/lib`. This should replace the library files originally located there. Back up the original files if needed.

All the instructions are also described in the `README` files of each directory of the artifact.

A.3.1 Installation

The experimental evaluation requires the following software.

(I1): pgmpy: https://pgmpy.org

(I2): NetworKit: https://networkit.github.io

(I3): angr: https://docs.angr.io

(I4): Visual Studio Build Tools: https://visualstudio.microsoft.com/downloads/?q=build+tools

A.3.2 Basic Test

We prepared a simple functionality test inside the `function-identification` folder of the artifact. The execution of command `python -m run_tests` from the directory `function-identification/tests` performs construction of the call graph on a test code and checks its values. The test should end within seconds and no assertions should be raised.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The utilization of probabilistic inference in our system significantly reduces the number of functions to be analyzed manually. The average rank of the genuine integrity protection function we seek is 1.56 as illustrated in Table 4 of the paper. This is evaluated in **(E1)**.

(C2): Our system detects a total of 34 mismatches from the 3 vendors we test. 29 of them are actual bugs including those that can lead to NAS AKA bypass. This is evaluated in **(E2)** and the results are summarized in Table 5 of the paper.

(C3): Our system complements dynamic testing in terms of completeness for analyzing integrity protection. Compared to DoLTEst and DIKEUE, which are previous works that use dynamic testing for analyzing integrity protection, our system covers more types of integrity protection bugs. This is evaluated with the same results of **(C2)** and the comparison results are illustrated in Table 6 of the paper.

A.4.2 Experiments

(E1): Identifying Integrity Protection [5 human-minute + 30 compute-minute] for each image: Ranks possible integrity protection functions in the target image.

Preparation: Follow the steps in Section A.3 to prepare the .idb files for analysis. To test with a different probability parameter value, change the `PROBABILITY_PARAMETER` value in the `utils.py` file under `function-identification/scripts/analyses`.

Execution: Load the `identify_integrity_function.py` file in `function-identification/scripts` to the .idb file of the target image.

Results: The results of the analysis should be written in the `results.txt` file of the target image's folder. A list of functions should be under the line written `Integrity Function Probability`. Starting with the function with the highest rank, the address of the function and probability is listed. The address of the genuine integrity protection function (reference result) is in the `symbolic-execution/config_firmware.yaml` file written as `integrity_func`. The rank should be the same as in Table 4 in the paper. Probability values may slightly differ by the number of functions identified. The time consumed for each step is also at the end of the file. We repeated the experiment several times while removing the cache every iteration and recorded the average in Table 7 of the paper.

(E2): Symbolic Execution [5 human-minute + 5 compute-minute] for each image: Finds mismatches with the specification in the integrity protection functions.

Preparation: Additional information about the image earned by manual analysis is required to be written in `symbolic-execution/config_firmware.py`. However, those for the provided images are already written down. Therefore, there is no preparation required for the evaluators to process.

Execution: Run the command `python analyze_base.py -fn {name_of_target}` in the `symbolic-execution` directory for each image.

Results: The results will be created under the `symbolic-execution/results/{name_of_target}` folder with the current time as the file name. The `Errored Results` and `Errored States` indicate the mismatches found by the system. The list under `Errored Results` consists of `[security state, security header, protocol discriminator, message type, reason_why_it_is_errored]` and the list under `Errored States` just indicates the reason

it is errored. We gathered the mismatches by the vendor and Table 5 in the paper indicates the results.

A.5 Notes on Reusability

To use our system on baseband firmware other than those provided in the artifact, mainly 2 steps would be required.

(S1): Based on whether our current implementation supports the vendor, a vendor-specific module might be needed to be written. The instructions for writing the module are well specified in the `README` files in the artifact.

(S2): Collect firmware-specific information such as the addresses of the security state, functions to skip, and so on.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.