USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

NASS: Fuzzing All Native Android System Services with Interface Awareness and Coverage

Philipp Mao, Marcel Busch, and Mathias Payer, *EPFL*

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

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: NASS: Fuzzing All Native Android System Services with Interface Awareness and Coverage

Philipp Mao   Marcel Busch   Mathias Payer
EPFL, Lausanne, Switzerland

A Artifact Appendix

A.1 Abstract

The artifact contains NASS’s source code which can be used to fuzz native system services on COTS devices. Futhermore the artifact contains scripts to run various aspects of the evaluation.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

NASS is designed to fuzz native system services on COTS devices. There is a danger of damaging the device, forcing a manual reset or making the phone unusable. For the artifact evaluation, all experiments running on real phones will be run on our testing devices exposed over ADB via port forwarding. NASS does not pose a risk to the host system it is run on.

A.2.2 How to access

The artifact is publicly available at https://doi.org/10.5281/zenodo.15577630 and https://github.com/HexHive/NASS.

A.2.3 Hardware dependencies

NASS either runs against an arm64 COTS Android device or an emulator. To evaluate NASS against FANS, an arm64 server is required in order to run the emulator with kvm support. Our experiments on COTS devices rely on the following phones:

Xiaomi Redmi Note 13
Google Pixel 9
Samsung S23
OnePlus 12R
Infinix X670

Note that for the artifact evaluation, we provide access to these phones remotly.

NASS’s host component should run on any Linux system. We use an Ubuntu 24.04 docker container.

A.2.4 Software dependencies

We ship a dockerfile in our repository which sets up all the software dependencies.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

To counter the hardware dependencies on an arm64 linux server and COTS Android devices, we provide access to an arm64 and x86 server. The access credentials are communicated in the `Artifact access` field on hotcrp.

The arm64 server is already setup with the emulator to reproduce the FANS evaluation (paragraphs with **FANS**). Access this server with `ssh -p 2222 ae@65.108.89.50` The x86 server has ADB access to our COTS testing devices, which we are forwarding from our local desktop. (paragraphs with **COTS**). Access this server with `ssh ae@65.108.89.50` See Figure 1 for the overview of the evaluation setup.

For both servers clone the NASS repository (if the folder is not present with)

`git clone https://github.com/HexHive/NASS`

Then navigate to the `./NASS` folder. Run the `./setup.sh` script. The script will setup NASS by building the docker. Then use `./run.sh` to spawn a shell in the NASS docker container.

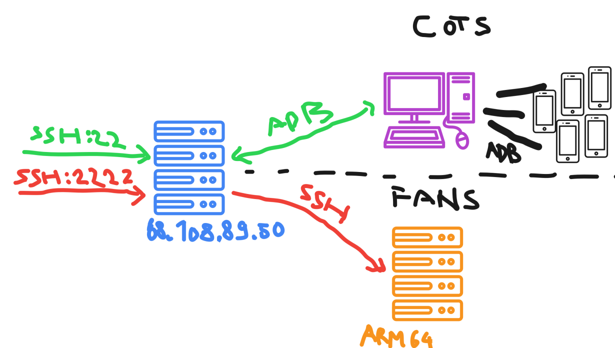

Figure 1: Overview of the evaluation setup.

A.3.2 Basic Test

All the steps here assume that you have logged in to the server, that `setup.sh` has finished successfully and the commands are inside of the NASS docker container.

COTS (ssh ae@65.108.89.50) Run the following command to list the devices connected to the server over ADB:

```
adb devices
```

The output should be the following:

```
List of devices attached
089092526K000893 device
47030DLAQ0012N device
RZCX312P76A device
a497c295 device
bai7gujvtchqeaus device
```

If this is not the case please let us know via hotcrp, it is possible the devices have shut down or the tunneling of ADB connections broke.

Assuming the five devices are present, run the following command which will fuzz a service on the Pixel 9 for 2 minutes.

```
./eval/cots/test_fuzz.sh
```

The output should contain the libfuzzer log lines akin to:

```
[LIBFUZZ] INFO: seed corpus: files: 1 min: 8b max: 8b total: 8b rss: 28Mb
[LIBFUZZ] #2 INITED ft: 42 corp: 1/8b exec/s: 0 rss: 28Mb
[LIBFUZZ] #4096 pulse ft: 42 corp: 1/8b lim: 4096 exec/s: 2048 rss: 29Mb
[LIBFUZZ] #8192 pulse ft: 42 corp: 1/8b lim: 4096 exec/s: 1638 rss: 29Mb
==========================
[ORC][47030DLAQ0012N] fuzzing iteration: 1 time: 62.078776836395264, crashes: 0, device borked: 0
[LIBFUZZ] #12486 NEW ft: 56 corp: 2/16b lim: 4096 exec/s: 1560 rss: 29Mb L: 8/8 MS: 1 ChangeCMd—
[LIBFUZZ] #16384 pulse ft: 56 corp: 2/16b lim: 4096 exec/s: 1489 rss: 29Mb
```

If this is displayed NASS was able to fuzz the `hardware.google.ril_ext.IRilExt/slot2` service on the Pixel 9.

FANS (ssh -p 2222 ae@65.108.89.50) Run the following command to fuzz a service on the arm64 emulator for 2 minutes.

```
./eval/fans/test_fuzz.sh
```

The output should contain the libfuzzer log akin to:

```
[ORC][emulator—5554] fuzzing iteration: 3 time: 24.192278385162354, crashes: 0, device borked: 0
[LIBFUZZ] #8192 pulse ft: 9 corp: 1/8b lim: 4096 exec/s: 372 rss: 28Mb
[LIBFUZZ] #9181 NEW ft: 70 corp: 2/16b lim: 4096 exec/s: 382 rss: 28Mb L: 8/8 MS: 1 ChangeCMd—
==========================
[ORC][emulator—5554] fuzzing iteration: 4 time: 30.4282705783844, crashes: 0, device borked: 0
[LIBFUZZ] #9245 NEW ft: 71 corp: 3/61b lim: 4096 exec/s: 385 rss: 28Mb L: 45/45 MS:
InsertEntry—
[LIBFUZZ] #9288 REDUCE ft: 71 corp: 3/40b lim: 4096 exec/s: 371 rss: 28Mb L: 24/24 MS: 1 InsertEntry—
[LIBFUZZ] #9295 REDUCE ft: 71 corp: 3/36b lim: 4096 exec/s: 371 rss: 28Mb L: 20/20 MS: 1 InsertEntry—
[LIBFUZZ] #9411 REDUCE ft: 71 corp: 3/24b lim: 4096 exec/s: 376 rss: 28Mb L: 8/8 MS: 1 DeleteEntry—
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Proprietary native services make up a large part of system services running on COTS devices (Table 1).

(C2): NASS finds bugs in proprietary services running on COTS devices (Table 5 and Table 6).

(C3): COTS services largely adhere to RPC design principles.

(C4): DGIE extracts accurate interface definitions from native services (Table 3).

(C5): NASS achieves parity with FANS (Figure 3, Table 4, Figure 4).

A.4.2 Experiments

All of the described commands assume that `setup.sh` been run on both servers and basic functionality test passed. We recommend to start a tmux session after logging in so longer-running commands are not terminated if the SSH connection drops. Simply run `tmux` after logging in over SSH. Furthermore all commands should be run inside of the NASS docker. To start the container use the `./run.sh` script. The following information including all the commands that need to be run can also be found in the `README.md` file.

COTS (ssh ae@65.108.89.50)

(E1): [Numbers of prop. native services, **C1**] [5 human-minutes + 15 compute-minutes]: Reproduce the numbers on proprietary native services on COTS devices from Table 1.

Preparation: Login to the COTS x86 server (ssh ae@65.108.89.50) and start the docker with `./run.sh`.

Execution: Run:

```
./eval/cots/native-service-stats.sh
```

Results: After the script has finished running, the numbers for each COTS device plus cumulative numbers are printed. These numbers should show that native services make up a large part of the system services running on COTS devices (**C1**) and be consistent with the numbers in Table 1. Note that the numbers do not match exactly since the firmware of the COTS devices has been updated since the evaluation was run for the paper.

(E2): [Fuzzing prop. native services, **C2**] [10 human-minutes + 3 compute-hours] Run NASS against proprietary native services on COTS devices.

Preparation: Login to the x86 server (ssh ae@65.108.89.50) and start the docker with `./run.sh`.

Execution: The provided script takes care of interface extraction, fuzzing and crash reproduction for a subset of native services. The script expects a device id. List these with `adb devices` then execute the run script with your chosen device id. Note it should be enough to execute the script once for a given device. For example, run:

```
./eval/cots/run-fuzz.sh 47030DLAQ0012N
```

to fuzz services on the Pixel 9.

Results: After the line `++EVAL++: analyzing crashes` the script will print the fuzzing output directories along with the reproduced crashes. Lines

starting with ========... mark the fuzzed service. If no reproducible crashes were found `NO CRASHES REPRODUCED` is printed. If crashes were reproduced, the following line is printed `@@@@@@@@@@REPRODUCED AND DEDUPLICATED@@@@@@@@@@@@@@@@@@@@@@`, followed by the `output-path` and the abbreviated crash cause. After fuzzing the Pixel 9 we expect two crashes to have been found, an abort due to fortify and a null pointer dererefence. The `output-path` contains the results of the fuzzing campaigns, including the seed corpus, crashes etc. For reproduced crashes the full crashlog can be found in the `[output-folder]/reproduced/[crash-*]/crashlog.txt` folder.

The output of this script, ran across all proprietary native services on the five devices was used to produce Table 5 and Table 6. In the scope of the artifact evaluation we believe that demonstrating NASS' functionality on a subset of services is sufficent to reproduce this experiment.

(E3): [COTS service compliance, **C3**] [10 human-minutes + 1 compute-minutes] Double check the manual ground truth analysis on COTS service compliance with the three RPC design principles.

Preparation: Login to the x86 server (ssh ae@65.108.89.50) and start the docker with `./run.sh`.

Execution: The provided scripts aggregates the numbers from the manually analyzed ground truth (stored in `./eval/cots/dgie_eval.csv`). Run:

```
python3 ./eval/cots/dgie_cots.py
```

Results: The script should print the numbers from Table 7. The csv contains the result of the manual analysis of compliance of COTS services with RPC design principles. The columns Ab/St denote if the service is compliant with RPC design principles "Abstraction of IPC binding code" / "Standard Deserialization Routines". Filling these columns requires manually analyzing the services: The service binary is decompiled and the entry point function reverse engineered. We believe that reproducing the csv in this way is outside of the scope of this artifact evaluation. All the data needed to reproduce the numbers, the service binaries along with the offset to the entry point function, are part of the artifact stored at `eval/cots/dgie_cots_eval`.

FANS (ssh -p 2222 ae@65.108.89.50)

(E4): [DGIE interface extraction, **C3**] [15 human-minutes + 2 compute-hours]: The evaluation script first extracts the interface for the FANS evaluation services using DGIE and then collects the results along with the results from the manual ground truth and captured RPC requests to reproduce Table 3.

Preparation: Login to the arm64 server (ssh -p 2222 ae@65.108.89.50), start a tmux session, and then start the NASS docker with `./run.sh`.

Execution: Run: `./eval/fans/ground-truth.sh`

Results: After the script has finished, the part of Table 3 that is automatically generated is printed. Check consistency of the script output with the numbers in Table 3. The column `#NASS Extr. RPC. funcs.` was analyzed manually. To check that the interface extracted by DGIE matches the ground truth interface, you can manually inspect the extracted interface and compare it to the ground truth interface. Both these are stored in json format mapping command identifier to the sequence of deserializers.

To print the manually extracted ground truth for the `installd` service run

```
cat eval/fans/ground_truth/aarch64emu28/installd.json|jq
```

To print DGIE's automatically extracted interface run

```
cat targets/aarch64emu28/installd/preprocess/interface.json|jq
```

For the `installd` service the automatically extracted deserializers should match the manual ground truth ones. Note that the notitation of deserializers in the manual json does not exactly match the name from the preprocessing, see Table 1. The same step may be repeated if desired for the other FANS evaluation services.

Manual	DGIE
String16	STRING16UTF8
String16Vector	STRING16UTF8VECTOR
ByteVector	BYTEARRAY

Table 1: Mapping from manual interface deserializers to DGIE extracted.

(E5): [FANS fuzzing campaign **C4**] [10 human-minutes + 13 compute-hours] Run a fuzzing campaign on the FANS evaluation services with NASS, NASS (NI) and FANS.

Preparation: Login to the arm64 server (ssh -p 2222 ae@65.108.89.50), start a tmux session, and then start the NASS docker with `./run.sh`.

Execution: The provided script takes care of running a shortened (1 hour) fuzzing campaign and produces the coverage graphs. Run: `./eval/fans/run_eval.sh ae` to run the fuzzing campaign.

Results: After finishing the resulting coverage graph pdfs can be found in `./eval/fans/run_out/ae/`. Download these from the server with: `scp -r -P 2222 ae@65.108.89.50:/home/ae/NASS/eval/fans/run_out/ae_out .`

The y axis is the code coverage from 0% to 100%, the x-axis is the time. the resulting coverage graphs should look similar to the *first hour of the graphs in the paper*. We choose to run a shortened campaign due to the time required to run the full experiment (around 2 weeks).

A.5 Notes on Reusability

We provide documentation in the repository on how to adjust NASS to fuzz proprietary native system services on other devices.

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/.