USENIX'23 Artifact Appendix: FirmSolo: Enabling dynamic analysis of binary Linux-based IoT kernel modules

Ioannis Angelakopoulos, Gianluca Stringhini, and Manuel Egele
Boston University
{jaggel, gian, megele}@bu.edu

A Artifact Appendix

A.1 Abstract

Our artifacts include the source code and instructions about how to install the FirmSolo prototype and use it to analyze the binary kernel modules of IoT firmware images. We provide two examples of firmware images that can be analyzed with FirmSolo. We also packaged FirmSolo within a docker image along with the two downstream analysis systems (i.e., Firmadyne and TriforceAFL) that we used to demonstrate FirmSolo's utility. The docker image can run on any system that has docker installed.

In this appendix, we describe the steps to analyze a sample firmware image using FirmSolo. The analysis process involves extracting metadata information from the kernel modules within the firmware image, reverse engineering the original firmware kernel and building a custom kernel capable of loading said modules. Finally, we demonstrate how downstream analysis tools can take advantage of FirmSolo to analyze the kernel modules within firmware images for bugs and vulnerabilities.

A.2 Description & Requirements

A.2.1 How to access

You can access the artifacts at https://github.com/BUseclab/FirmSolo/tree/v1.0.0

A.2.2 Hardware dependencies

None

A.2.3 Software dependencies

Python, Docker and Java (for Ghidra).

A.2.4 Benchmarks

For the artifact evaluation we use an example Netgear firmware image as a benchmark ($id = 1$).

A.3 Set-up

A.3.1 Installation

Using the Docker:

Download the docker image from:
https://doi.org/10.5281/zenodo.7865451

Install the docker image:

```
docker load < firmsolo.tar.gz
git clone https://github.com/BUseclab/FirmSolo.git
cd FirmSolo
docker build -t firmsolo .
```

Spawn a docker container:

```
docker run -v $(pwd):/output --rm -it
--privileged firmsolo /bin/bash
```

Manual Installation:

To manually install and run FirmSolo you first need to install these dependencies:

```
 sudo apt-get install build-essential
zlib1g-dev pkg-config libglib2.0-dev
binutils-dev libboost-all-dev autoconf libtool
libssl-dev libpixman-1-dev libpython3-dev
python3-pip python3-capstone python-is-python3
virtualenv sudo gcc make g++ python3 python2
flex bison dwarves kmod universal-ctags fdisk
fakeroot git dmsetup kpartx netcat-openbsd
nmap python3-psycopg2 snmp uml-utilities
util-linux vlan busybox-static postgresql wget
cscope qemu qemu-system-arm qemu-system-mips
qemu-system-mipsel qemu-utils
```

Install these python packages:

```
pip3 install ply anytree sympy requests
pexpect scipy
```

Within the FirmSolo installation directory run:

```
git submodule init
git submodule update
```

Install Ghidra:

Follow instuctions in https://ghidra-sre.org/InstallationGuide.html

Download TriforceAFL:

```
git clone https://github.com/BUseclab/TriforceAFL.git
cd TriforceAFL && make
```

Download TriforceLinuxSyscallFuzzer:

```
git clone https://github.com/BUseclab/Triforce-LinuxSyscallFuzzer.git
```

```
cd TriforceLinuxSyscallFuzzer && ./compile_harnesses.sh
```

Note: The `compile_harnesses.sh` script will make use of legacy (and unavailable) compiler toolchains that are currently only installed within the Docker image.

Download Firmadyne:

```
git clone --recursive https://github.com/BUseclab/firmadyne.git
```

Download the buildroot filesystems:

https://drive.google.com/file/d/11GiU8N1U4Nkhv-kurkoGgwmp38CM_Umg/view?usp=share_link and download the buildroot_fs.tar.gz file within FirmSolo's installation directory.

Execute:

```
tar xvf builroot_fs.tar.gz
```

Finally, specify the toolchain to be used by FirmSolo. Go into the installation directory of FirmSolo and edit the `custom_utils.py` script. Within the `get_toolchain` function edit the `cross` variable with the path(s) to your toolchain(s).

A.3.2 Basic Test

Our basic test for FirmSolo includes statically analyzing the kernel modules within a firmware image to extract metadata about the original firmware kernel. Please proceed as follows:

Spawn a docker container according to Section A.3.1 and run:

```
mkdir -p /output/images/
```

On your host download the images from this link: https://drive.google.com/file/d/1xzdTAz3PexQD8YWWAg7KYyQ8dQiVTGiR/view?usp=share_link

Execute:

```
tar xvf examples.tar.gz
cp -r ./examples/* <work_dir>/images/
```

The `work_dir` is your work directory (e.g., `./`)

Then inside your container execute:

```
cd /FirmSolo
python3 firmsolo.py -i 1 -s 1
ls /output/Image_Info/
```

After running the `ls` command you should see a file named `1.pkl` within the `/output/Image_Info/` directory.

A.4 Evaluation workflow

A.4.1 Major Claims

FirmSolo is a framework that exposes Linux-based binary IoT kernel modules to downstream analysis. Below we list and prove the claims related to the evaluation of our artifact.

(C1): *FirmSolo reverse engineers the original kernel of a firmware image and builds a new kernel supported by QEMU that can load the kernel modules within the firmware image. This claim is proven by experiment (E1), described in Section 5.2, Table 2 and Figure 2 in our paper.*

(C2): *Downstream analysis systems can use FirmSolo to analyze binary firmware kernel modules for bugs and vulnerabilities. This claim is proven in Experiments (E2) and (E3), described in Section 5.4, Table 5 and Table 6 in our paper.*

A.4.2 Experiments

We assume that you use the docker image to perform the artifact evaluation.

(E1): *[Reverse Engineering] [5 human-minutes + 10 compute-minutes + 5GB disk]: In this experiment FirmSolo extracts metadata from the binary kernel modules of a firmware image, reverse engineers the original firmware kernel and builds a new kernel capable of loading the kernel modules within the firmware image.*

Preparation: *Copy the extracted file-system and kernel of the target firmware image in the work directory.*

Execution: *After you install and connect to the docker container as described in Section A.3.1, proceed to analyze an example firmware image:*

Within the docker execute:

```
mkdir -p /output/images/
```

On your host download the images from this link (if you have not implemented the basic test): https://drive.google.com/file/d/1xzdTAz3PexQD8YWWAg7KYyQ8dQiVTGiR/view?usp=share_link

Execute:

```
tar xvf examples.tar.gz
cp -r ./examples/* <work_dir>/images/
```

The `work_dir` is your work directory (e.g., `./`)

To analyze image 1 with FirmSolo, inside your container execute:

```
cd /FirmSolo
python3 firmsolo.py -i 1 -a
```

FirmSolo will analyze firmware image 1, reverse engineer the original firmware kernel and build a new kernel that is capable of loading the kernel modules within image 1. FirmSolo will also find which kernel modules can actually load and also which kernel modules crash during emulation (if any). If any kernel modules crashed during emulation, FirmSolo will try to address the error by running stage 2c.

Results: *To get information about the analyzed image 1, such as the kernel modules within the firmware image, the kernel modules that loaded successfully using the FirmSolo kernel, the kernel modules that crashed during emulation and the kernel module substitutions implemented by FirmSolo, run this command:*

```
python3 firmsolo.py -i 1 -d
```

In the output you should see:

```
Image: 1 Total Modules: 16 Loaded Modules:
5 Crashing Modules: 0 Substitutions: 0
```

along with specific information about which modules were successfully loaded, crashed, and substituted.

Note: Depending on the metadata information extracted/processed in this step (e.g., kernel symbols) this step can take longer. However, FirmSolo caches data about each kernel version used by the firmware images it analyzes, which renders future runs faster.

(E2): *[TriforceAFL] [1 human-minute + 1.5 compute-hour]: In this experiment you use the TriforceAFL kernel fuzzer to analyze the kernel modules within the target firmware image.*

Preparation: *None*

Execution: *Setup TriforceAFL with these commands:*

```
echo core >/proc/sys/kernel/core_pattern
cd /sys/devices/system/cpu
echo performance | tee
cpu*/cpufreq/scaling_governor
```

These commands are needed by AFL for improving performance. To fuzz the kernel modules within image 1 for 30 minutes each run:

```
python3 ./triforceafl/triforce_run.py -i 1
-t 30m
```

The `triforce_run.py` script will analyze the binary kernel modules within image 1 which expose an IOCTL interface. The script will find potential IOCTL command numbers that can be used to access these IOCTL interfaces and will use the command numbers found as seeds for TriforceAFL. Image 1 has two kernel modules that can be fuzzed (*acos_nat.ko* and *ipv6_spi.ko*). The kernel module *acos_nat.ko* exposes two IOCTL interfaces and each will be fuzzed separately. Thus the total fuzzing time for both kernel modules will be around 90 minutes.

Results: *The fuzzing results will be available in the `/output/Fuzz_Results_Cur/1` directory. To be able to quickly test for a crash found by the fuzzer run this command:*

```
python3 ./triforceafl/get_fuzzing_cmd.py 1
```

If the fuzzer triggered any crashes for any of the IOCTL interfaces fuzzed, the `get_fuzzing_cmd.py` script will output commands that can be copy/pasted into the terminal and executed to quickly test a crash. The commands will be available under the `CRASHES:` section (for each IOCTL interface) else this section will be blank.

(E3): *[Firmadyne] [1 human-minute + 30 compute-minutes]: In this experiment you use the Firmadyne dynamic analysis system to analyze the kernel modules within the target firmware image.*

Preparation: *None*

Execution: *Run the Firmadyne analysis for image 1 as follows:*

```
cd /firmadyne && ./experiment.sh 1
```

The `experiment.sh` script will run a full analysis with Firmadyne; creating a file-system, detecting a network configuration and testing the firmware kernel modules of image 1 against exploits from ExploitDB and the bugs found by TriforceAFL as explained in our paper.

Results: *The results will be available in the `/output/firmadyne_results/1` directory.*

To check if any of the exploits from ExploitDB and the TriforceAFL bugs triggered a crash, manually inspect the serial logs under `/output/firmadyne_results/1/[remote,local]/` and `/output/firmadyne_results/1/afl/`, respectively.

The `qemu.final.serial.log_2694_ipv6_spi_1` file in `/output/firmadyne_results/1/afl/` should contain an “Oops” message. It might be the case though that the crash message is not present because the kernel hangs before printing it. In this case you may need to re-run the analysis.

A.5 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/.