Verifying the Verifier: eBPF Range Analysis Verification

Abstract

Our paper proposes an automated method to check the correctness of range analysis used in the Linux Kernel's eBPF verifier. In this artifact, we provide our software that (a) automatically extracts the abstract semantics of the range analysis from the C code, (b) generates the verification conditions for the soundness of the range analysis and checks the verifier's range analysis for soundness, and (c) synthesizes an eBPF program that demonstrates the mismatch between the abstract and the concrete semantics in the case where the range analysis fails to meet the soundness condition.

Availability

This artifact is publicly available at zenodo (doi: 10.5281/zenodo.7877222), and github (https://github.com/bpfverif/ebpf-range-analysis-verification-cav23). You can read this readme with markdown highlighting directly on github.

Functionality

The entire set of experiments can take up to 6 days to complete. This artifact contains all necessary tools and software needed to replicate the full set of results from the paper from scratch. To expedite the evaluation, we provide this section that skips some time-consuming steps while still reproducing the results from the paper.

Reusability

Our artifact is open source with the MIT License. This artifact contains a Docker image which includes all the software and dependencies. However, our software is portable to different environments and platforms and we also list the dependencies and libraries. We explain the structure of our code and how to extend it.

Prerequisites to run the artifact.

1. Install Docker if not already installed by following the documentation here. You might need to follow the post installation steps for managing docker as a non-root user here.
2. Install Virtual Box if not already installed by downloading from here.

`Note:` If your goal is solely to reproduce the results from the paper, you can proceed directly to the section titled Reproducing Results from the Paper. However, if you are interested in learning how to utilize our software, please continue reading below.

Claims to validate/reproduce.

(1) Our software can successfully start from the Linux kernel verifier's C source code and produce SMT that encodes the semantics of the Linux kernel's C code.

(2.1) Our software is able to verify the kernel's range analysis using our `gen` and `sro` verification conditions.

(2.2) Our software can synthesize proof-of-concept BPF programs demonstrate a mismatch between the concrete and abstract semantics.

(3) Our synthesized proof-of-concept BPF programs manifest unsound behavior in a real Linux kernel.

Artifact functionality, system requirements and known issues.

In the paper, we report the results of the verification and synthesis in Fig. 5(a) and 5(b) respectively, for several kernel versions. In this artifact, we demonstrate our software for a specific kernel version, 5.9. It is straightforward to use it for other kernel versions.

We have tested this artifact, including the Docker image and the Virtual Box appliance, on a machine with 4 physical cores running at 2.8 GHz and 16 GB of memory. We have tested the artifact on machines running Linux and Windows operating systems, both using x86_64 architecture, and we have no known issues to report.

Dependencies

Our software has the following dependencies

```
clang-12
llvm-12
llvm-12-tools
python3
```

```
make
cmake
libjsoncpp-dev
libz3-dev
```

It also depends on the following python packages:

```
packaging
prettytable
termcolor
toml
z3-solver
```

(1.) Automatically extracting the semantics of the Linux kernel's C code to SMT (30 minutes)

Here, we demonstrate how our tool can be used to *automatically* extract the semantics of the Linux Kernel verifier's C code as described in our paper (§5). Our tool produces the first-order logic formula (in SMT-LIB format) for the abstract semantics defined in Linux Kernel for each eBPF instruction.

Load and run the docker image

```
docker load < cav23-artifact-docker.tar
docker run -it cav23-artifact:publish
```

Clone the Linux git repository (15 minutes)

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git /home/linux-stable
```

Create the output directory and run the llvm-to-smt tool (15 minutes)

```
mkdir /home/cav23-artifact/bpf-encodings-5.9
cd /home/cav23-artifact/llvm-to-smt
python3 generate_encodings.py --kernver 5.9 --kernbasedir /home/linux-stable --outdir /home/cav23-
artifact/bpf-encodings-5.9
```

Expected Result

```
Log file: /home/cav23-artifact/bpf-encodings-5.9/log_21_58_27_04_2023.log
Log error file: /home/cav23-artifact/bpf-encodings-5.9/log_err_21_58_27_04_2023.log
Change to kernel directory: /home/linux-stable ... done
Checkout kernel version v5.9 ... done
Run make config and edit BPF flags ... done
Extract compile flags for current kernel version ... done
Edit tnum.c and verifier.c to add wrappers ... done
Compile verifier.c and tnum.c ... done
Link verifier.ll and tnum.ll to single verifier.ll ... done
Getting encoding for BPF_ADD ... done
Getting encoding for BPF_SUB ... done
Getting encoding for BPF_OR ... done
Getting encoding for BPF_AND ... done
Getting encoding for BPF_LSH ... done
.
.
.
Getting encoding for BPF_SYNC ... done
```

Explanation

Our automatic encoder produces an SMT-LIB (`.smt2`) file for each eBPF instruction in the output directory (`/home/bpf-encodings-5.9`), that captures the Linux Kernel's abstract semantics for the instruction. We now have the semantics of 36 abstract operators corresponding to 36 eBPF instructions.

```
root@847d5c0f8828:/home/cav23-artifact/llvm-to-smt# ls -1 /home/cav23-artifact/bpf-encodings-5.9/*.smt2
/home/bpf-encodings-5.9/BPF_ADD.smt2
/home/bpf-encodings-5.9/BPF_ADD_32.smt2
/home/bpf-encodings-5.9/BPF_AND.smt2
.
.
.
```

(2.1 & 2.2) Verification and POC synthesis for eBPF range analysis (~13 hours)

We now check the correctness of the 36 abstract operators using our verification conditions `gen` (§4.1) and `sro` (§4.2). When our soundness checks fail, we synthesize proof-of-concept (PoC) programs that demonstrate the mismatch between abstract values maintained by the verifier and the values in a concrete execution of the eBPF program. Note that the synthesized PoC programs will be for demonstrative purposes only. Constructing a full eBPF program from our generated POCs requires some manual effort. For the review, we will forgo this step. In part 3 of the artifact, we provide full eBPF programs constructed from the demonstrative examples that manifest unsound behaviors in an actual kernel.

Run the script to perform the verification and synthesis

The script uses the encodings we previously generated, present in `/home/cav23-artifact/bpf-encodings-5.9`.

```
cd /home/cav23-artifact/bpf-verification
mkdir results/
cd src/
python3 bpf_alu_jmp_synthesis.py --kernver 5.9 --encodings_path /home/cav23-artifact/bpf-encodings-5.9
```

Expected result

The expected output should be similar to the one below. Note that the order of instructions in the verification part and the order of synthesized programs in the synthesis part might differ. Each row in the tables however, should be the same.

```
------------------------------------------------------------
               2.1(a) Executing gen verification
------------------------------------------------------------

1/15 Verifying BPF_JNE ...  Done.
2/15 Verifying BPF_JEQ ...  Done.
3/15 Verifying BPF_JLT ...  Done.
4/15 Verifying BPF_OR_32 ...  Done.
5/15 Verifying BPF_AND ...  Done.
6/15 Verifying BPF_JGE ...  Done.
7/15 Verifying BPF_JGT ...  Done.
8/15 Verifying BPF_JSGT ...  Done.
9/15 Verifying BPF_OR ...  Done.
10/15 Verifying BPF_SUB ...  Done.
11/15 Verifying BPF_AND_32 ...  Done.
12/15 Verifying BPF_JLE ...  Done.
13/15 Verifying BPF_JSGE ...  Done.
14/15 Verifying BPF_JSLE ...  Done.
15/15 Verifying BPF_JSLT ...  Done.
Gen Verification Complete
+-------------+--------+-----+-----+------+-----+-----+--------------------------+
| Instruction | Sound? | u64 | s64 | tnum | u32 | s32 | Execution time (seconds) |
+-------------+--------+-----+-----+------+-----+-----+--------------------------+
|   BPF_JNE   |   ✘    |  ✘  |  ✘  |  ✘   |  ✘  |  ✘  |          192.43          |
|   BPF_JEQ   |   ✘    |  ✘  |  ✘  |  ✘   |  ✘  |  ✘  |          89.32           |
|   BPF_JLT   |   ✘    |  ✘  |  ✘  |  ✘   |  ✘  |  ✘  |          55.04           |
|  BPF_OR_32  |   ✘    |  ✘  |  ✘  |  ✓   |  ✘  |  ✘  |           5.18           |
|   BPF_AND   |   ✘    |  ✓  |  ✘  |  ✓   |  ✘  |  ✘  |           9.53           |
|   BPF_JGE   |   ✘    |  ✘  |  ✘  |  ✘   |  ✘  |  ✘  |          33.15           |
|   BPF_JGT   |   ✘    |  ✘  |  ✘  |  ✘   |  ✘  |  ✘  |          42.89           |
```

```
|   BPF_JSGT  |   ✘    |  ✘  |  ✘  |  ✘   |  ✘  |  ✘  |           28.4           |
|    BPF_OR   |   ✘    |  ✓  |  ✘  |  ✓   |  ✘  |  ✘  |           9.21           |
|   BPF_SUB   |   ✘    |  ✘  |  ✘  |  ✘   |  ✓  |  ✓  |          116.87          |
|  BPF_AND_32 |   ✘    |  ✘  |  ✘  |  ✓   |  ✘  |  ✘  |           5.28           |
|   BPF_JLE   |   ✘    |  ✘  |  ✘  |  ✘   |  ✘  |  ✘  |          62.81           |
|   BPF_JSGE  |   ✘    |  ✘  |  ✘  |  ✘   |  ✘  |  ✘  |          38.35           |
|   BPF_JSLE  |   ✘    |  ✘  |  ✘  |  ✘   |  ✘  |  ✘  |          32.33           |
|   BPF_JSLT  |   ✘    |  ✘  |  ✘  |  ✘   |  ✘  |  ✘  |          50.91           |
+-------------+--------+-----+-----+------+-----+-----+--------------------------+

------------------------------------------------------------
                2.1(b) Executing sro verification
------------------------------------------------------------

1/15 Verifying BPF_JNE ...  Done.
2/15 Verifying BPF_JEQ ...  Done.
3/15 Verifying BPF_JLT ...  Done.
4/15 Verifying BPF_OR_32 ...  Done.
5/15 Verifying BPF_JGE ...  Done.
6/15 Verifying BPF_JGT ...  Done.
7/15 Verifying BPF_JSGT ...  Done.
8/15 Verifying BPF_OR ...  Done.
9/15 Verifying BPF_SUB ...  Done.
10/15 Verifying BPF_JSLE ...  Done.
11/15 Verifying BPF_AND_32 ...  Done.
12/15 Verifying BPF_JLE ...  Done.
13/15 Verifying BPF_JSGE ...  Done.
14/15 Verifying BPF_AND ...  Done.
15/15 Verifying BPF_JSLT ...  Done.
SRO Verification Complete
+-------------+--------+-----+-----+------+-----+-----+--------------------------+
| Instruction | Sound? | u64 | s64 | tnum | u32 | s32 | Execution time (seconds) |
+-------------+--------+-----+-----+------+-----+-----+--------------------------+
|   BPF_JNE   |   ✘    |  ✘  |  ✘  |  ✘   |  ✘  |  ✘  |          203.43          |
|   BPF_JEQ   |   ✘    |  ✘  |  ✘  |  ✘   |  ✘  |  ✘  |         3911.46          |
|   BPF_JLT   |   ✘    |  ✘  |  ✘  |  ✘   |  ✘  |  ✘  |          74.94           |
|  BPF_OR_32  |   ✘    |  ✘  |  ✘  |  ✓   |  ✘  |  ✘  |          17.38           |
|   BPF_JGE   |   ✘    |  ✘  |  ✘  |  ✘   |  ✘  |  ✘  |          120.62          |
|   BPF_JGT   |   ✘    |  ✘  |  ✘  |  ✘   |  ✘  |  ✘  |          225.68          |
|   BPF_JSGT  |   ✘    |  ✘  |  ✘  |  ✘   |  ✘  |  ✘  |          114.93          |
|    BPF_OR   |   ✘    |  ✓  |  ✓  |  ✓   |  ✘  |  ✘  |          26.93           |
|   BPF_SUB   |   ✘    |  ✘  |  ✘  |  ✘   |  ✓  |  ✓  |          213.83          |
|   BPF_JSLE  |   ✘    |  ✘  |  ✘  |  ✘   |  ✘  |  ✘  |          79.36           |
|  BPF_AND_32 |   ✘    |  ✘  |  ✘  |  ✓   |  ✘  |  ✘  |          22.71           |
|   BPF_JLE   |   ✘    |  ✘  |  ✘  |  ✘   |  ✘  |  ✘  |          103.7           |
|   BPF_JSGE  |   ✘    |  ✘  |  ✘  |  ✘   |  ✘  |  ✘  |          95.13           |
|   BPF_AND   |   ✘    |  ✓  |  ✓  |  ✓   |  ✘  |  ✘  |          31.16           |
|   BPF_JSLT  |   ✘    |  ✘  |  ✘  |  ✘   |  ✘  |  ✘  |          71.92           |
+-------------+--------+-----+-----+------+-----+-----+--------------------------+


------------------------------------------------------------
                2.2 Generating POC for domain violations
------------------------------------------------------------


Synthesized program for BPF_JLT (signed_64). Instruction sequence: BPF_JLT
Synthesized program for BPF_JLT (Tnum). Instruction sequence: BPF_JLT
Synthesized program for BPF_JLT (unsigned_32). Instruction sequence: BPF_JLT
Synthesized program for BPF_JLT (signed_32). Instruction sequence: BPF_JLT
Synthesized program for BPF_JGE (signed_64). Instruction sequence: BPF_JGE
Synthesized program for BPF_JGE (Tnum). Instruction sequence: BPF_JGE
Synthesized program for BPF_JGE (unsigned_32). Instruction sequence: BPF_JGE
Synthesized program for BPF_JGE (signed_32). Instruction sequence: BPF_JGE
Synthesized program for BPF_JGT (signed_64). Instruction sequence: BPF_JGT
Synthesized program for BPF_JGT (Tnum). Instruction sequence: BPF_JGT
.
.
.
Synthesized program for BPF_OR_32 (unsigned_64). Instruction sequence: BPF_JSLE BPF_OR_32
Synthesized program for BPF_OR_32 (signed_64). Instruction sequence: BPF_JSLE BPF_OR_32
Synthesized program for BPF_OR_32 (unsigned_32). Instruction sequence: BPF_JSLE BPF_OR_32
```

```
Synthesized program for BPF_OR_32 (signed_32). Instruction sequence: BPF_JSLE BPF_OR_32
Synthesized program for BPF_OR (unsigned_32). Instruction sequence: BPF_JSLE BPF_OR
Synthesized program for BPF_OR (signed_32). Instruction sequence: BPF_JSLE BPF_OR

================================================================================

Verification Aggregate Statistics
+---------+------------+------------+-----------+-----------+-----------------+-----------------+
| KernVer | gen Sound? | sro Sound? | gen Viol. | sro Viol. | gen Unsound Ops | sro Unsound Ops |
+---------+------------+------------+-----------+-----------+-----------------+-----------------+
|  5.9    |     ✘      |     ✘      |    67     |    65     |       15        |      15         |
+---------+------------+------------+-----------+-----------+-----------------+-----------------+

Synthesis Aggregate Statistics
+---------+--------------+-----------------------+------------+------------+------------+
| KernVer | # Tot. Viol. | All POCs Synthesized? | Prog Len 1 | Prog Len 2 | Prog Len 3 |
+---------+--------------+-----------------------+------------+------------+------------+
|  5.9    |    65        |           ✓           |    39      |     26     |     0      |
+---------+--------------+-----------------------+------------+------------+------------+
```

Explanation

- The first part of this experiment `2.1(a) Executing gen verification`, corresponds to the verification on the set of eBPF instructions using our `gen` verification condition. The table at the end of this part denotes whether an instruction was deemed sound (✓) or unsound (✘), and which of the five abstract domains have been violated.
- The second part `2.1(b) EXECUTING sro VERIFICATION` performs verification on the eBPF instructions that were deemed unsound by the previous `gen` verification condition, using our `sro` verification condition. It produces a similar table as in the prior part.
- The third part `2.2 Generating POC for domain violations` performs synthesis. It attempts to generate proof-of-concept mini eBPF programs using instructions that were deemed unsound by both the `gen` and `sro` verification conditions from the previous steps. The programs manifest a mismatch between verifier's abstract values and the concrete execution, that is they demonstrate unsound behavior.
- Lastly, we provide two tables with aggregate statistics.
    - The first table `Verification Aggregate Statistics` shows aggregate statistics for the verification part of the experiment (2.1(a) and 2.1(b)). This table should match exactly with Fig 5(a) (row kernel version 5.9) from the paper.
    - The second table `Synthesis Aggregate Statistics` summarizes the total number of unsound instructions + domain pairs (i.e. the total number of violations). It also shows whether the synthesis was successful in producing a program for all the violations, as well as the respective program lengths. This table should match with Fig. 5(b) from the paper.

In general, each POC is documented in a separate log in `/home/cav23-artifact/bpf-verification/results`. We use these logs to manually craft a full eBPF program from the output of the synthesis process.

(3) Running synthesized eBPF programs in a real Linux Kernel (20 minutes)

In the previous experiment, we synthesized mini eBPF programs. These programs are utilized to create a complete eBPF program, which showcases a discrepancy between the abstract values maintained by the verifier and the actual execution of the eBPF program. Building a full eBPF program requires some manual effort. However, to simplify the review process, we offer pre-constructed eBPF programs that we have created using the output from the synthesis. These programs are available in our Virtual Box appliance named cav23-artifact-vm.ova. This appliance contains a virtual machine that operates on Ubuntu 20.04 and is equipped with Linux Kernel v5.9.

Import and start the virtual machine

- Open Virtual Box
- File > Import Appliance
- Browse and select the path to `cav23-artifact-vm.ova`
- You should have the `cav23-artifact-vm` virtual machine imported and ready the sidebar.
- Double-click on it or press "Start" to start the VM.
- If prompted, choose "Ubuntu" in the grub boot menu.
- The password for login is `cav23`.

Setup

Open a terminal. The directory `/home/cav23-reviewer/bpf_progs/` contains all the code necessary to run our synthesized eBPF programs. These programs are located in `/home/cav23-reviewer/bpf_progs/pocs`. The naming convention for these eBPF programs are `<kernel_version>_<ebpf_isntruction>_<domain_violated>.c`. Since our VM is installed with kernel version 5.9, we will only work with the `5.9*` files.

```
ls /home/cav23-reviewer/bpf_progs/pocs/*.c
5.5_jne_tnum_manfred.c  5.5_jsle_umin.c     5.7rc1_arsh32_u64.c
5.7rc1_xor32_u64.c      5.8_sub_s64.c       5.9_or_s32.c
5.5_jsgt32_tnum.c       5.7rc1_add32_u64.c  5.7rc1_jge32_u64.c
5.8_jsgt_s32.c          5.9_jsgt_s32.c      5.9_sub_s64.c
```

Run example eBPF program 1: a bug in 64-bit BPF_SUB

We will now run the `5.9_sub_s64.c` program. This program causes the verifier's range tracking to reach an invalid state. The verifier believes that the minimum possible value in a register `smin_value` is greater than the maximum value `smax_value`. This is clearly unsound.

```
cd /home/cav23-reviewer/bpf_progs/
cp pocs/5.9_sub_s64.c ./bpf_test.c
sudo make
./bpf_test
```

The output should be somewhat similar to the one below. Scroll down to the lines named `7`. `smin_value` is indeed greater than `smax_value`.

```
6: (1f) r1 -= r2
7: R1(id=0,smin_value=3550221988887957680,smax_value=3550221988887957679,umin_value=3550221988887957680,u
max_value=3550221988887957679,var_off=(0x3144ec1600000000; 0xffffffff),s32_min_value=-2147483648,s32_max_
value=2147483647,u32_min_value=0,u32_max_value=-1) R2(id=0,smin_value=-9223372036854775808,smax_value=922
3372036854775807,umin_value=0,umax_value=18446744073709551615,var_off=(0x0; 0xffffffffffffffff),s32_min_v
alue=-2147483648,s32_max_value=2147483647,u32_min_value=0,u32_max_value=-1) R10(id=0,smin_value=0,smax_va
lue=0,umin_value=0,umax_value=0,var_off=(0x0; 0x0),s32_min_value=0,s32_max_value=0,u32_min_value=0,u32_ma
x_value=0)
7: (b7) r0 = 0
```

Run example eBPF program 2: a bug in 32-bit BPF_OR

```
cd /home/cav23-reviewer/bpf_progs/
cp pocs/5.9_or_s32.c ./bpf_test.c
sudo make
./bpf_test
```

Similar to the above program, this program demonstrates that minimum possible value in a 32-bit sub register `s32_min_value` is greater than the maximum 32-bit value `s32_max_value`. This too, is clearly unsound. To see this, scroll down to the line named `14`.

```
13: (4f) r1 |= r4
14: R1(id=0,smin_value=-8952222646855008257,smax_value=9223372036854775807,umin_value=27
11493899999767551,umax_value=18446744073709551615,var_off=(0x3c350eeffffffff; 0xfc3caf110
0000000),s32_min_value=-1,s32_max_value=-2,u32_min_value=-1,u32_max_value=-2) R2(id=0,sm
in_value=1,smax_value=1,umin_value=1,umax_value=1,var_off=(0x1; 0x0),s32_min_value=1,s32
_max_value=1,u32_min_value=1,u32_max_value=1) R3(id=0,smin_value=-9223372036854775808,sm
ax_value=9223372036854775807,umin_value=0,umax_value=18446744073709551615,var_off=(0x0;
0xffffffffffffffff),s32_min_value=-2147483648,s32_max_value=2147483647,u32_min_value=0,u
32_max_value=-1) R4(id=0,smin_value=271149389999767551,smax_value=271149389999767551,umi
n_value=271149389999767551,umax_value=271149389999767551,var_off=(0x3c350eeffffffff; 0x0
),s32_min_value=-1,s32_max_value=-1,u32_min_value=-1,u32_max_value=-1) R10(id=0,smin_val
ue=0,smax_value=0,umin_value=0,umax_value=0,var_off=(0x0; 0x0),s32_min_value=0,s32_max_v
alue=0,u32_min_value=0,u32_max_value=0)
14: (b7) r0 = 1
```

Reproducing results from the paper

We now show how to quickly reproduce the results from the paper. The script `reproduce_results.py` achieves this by:

- Skipping the generation of the encoding of the abstract semantics of eBPF instructions directly from the Linux C source code. Instead, we provide our own encodings that we generated using our llvm-to-smt tool. These encodings are stored in the `/home/cav23-artifact/reproduce-results` directory.
- For each kernel version, storing a list of eBPF instructions whose abstract semantics are known to be *unsound*. E.g. for v5.9, 15 of the 36 instructions are known to be unsound. The script then runs the verification on this reduced list of instructions. to confirm that this list of

instructions are indeed unsound. These reduced list of instructions are stored in the script `reproduce_results.py` directly.
- Restricting the length of programs synthesized to 1 instruction. That is, the script only produces single instruction eBPF programs that manifest a mismatch between verifier's abstract values and the concrete execution.

Reproducing the table for a particular kernel version

We show how to reproduce the verification results from Fig 5(a) and the synthesis results from Fig 5(b), for a *particular* kernel version. Note that we recommend running a separate docker container for each kernel version if you want to reproduce the results for several kernel versions in parallel. For example, for reproducing the results related to kernel v5.13, do the following on your host machine to start a new container:

```
docker run -it cav23-artifact:publish
```

Then, within the container:

```
cd /home/cav23-artifact/reproduce-results
python3 reproduce_results.py --kernver 5.13
```

Similarly, to reproduce the results related to v5.7-rc1, open a new terminal on your host machine, and spin up a new container:

```
docker run -it cav23-artifact:publish
```

Then, within the container:

```
cd /home/cav23-artifact/reproduce-results
python3 reproduce_results.py --kernver 5.7-rc1
```

Reading the results

The output of `reproduce_results.py` contains two tables at the tail end. For example, the output for kernel v5.9 should be as follows:

```
.
.
.
Verification Aggregate Statistics
+---------+------------+------------+-----------+-----------+-----------------+-----------------+
| KernVer | gen Sound? | sro Sound? | gen Viol. | sro Viol. | gen Unsound Ops | sro Unsound Ops |
+---------+------------+------------+-----------+-----------+-----------------+-----------------+
|   5.9   |     ✘      |     ✘      |     67    |     65    |        15       |        15       |
+---------+------------+------------+-----------+-----------+-----------------+-----------------+

Synthesis Aggregate Statistics
+---------+--------------+-----------------------+------------+------------+------------+
| KernVer | # Tot. Viol. | All POCs Synthesized? | Prog Len 1 | Prog Len 2 | Prog Len 3 |
+---------+--------------+-----------------------+------------+------------+------------+
|   5.9   |      65      |           ✘           |     39     |     0      |     0      |
+---------+--------------+-----------------------+------------+------------+------------+
```

- The first table `Verification Aggregate Statistics` shows aggregate statistics for the verification part of the experiment. This table should match exactly with Fig 5(a) (row kernel version 5.9) from the paper.
- The second table `Synthesis Aggregate Statistics` summarizes the total number of unsound instructions + domain pairs (i.e. the total number of violations). It also shows whether the synthesis was successful in producing a program for all the violations, as well as the respective program lengths. This table should match with Fig. 5(b) from the paper.

  `Note`: Becase we are restricting the synthesized program lengths to 1, the columns for `Prog Len 2` and `Prog Len 3` will always be populated with 0.

Each row in Fig 5(a), and Fig. 5(b) can be similarly verified by checking the output of each instances of the `reproduce_results.py` script running in its own docker container.

Source code structure and extensibility

C to SMT

```
llvm-to-smt
├── llvm-passes
│   ├── ForceFunctionEarlyExit
│   ├── InlineFunctionCalls
│   ├── LLVMToSMT
│   ├── PromoteMemcpy
│   ├── RemoveFunctionCalls
├── generate_encodings.py
├── run_llvm_passes.py
└── wrappers.py
```

Our first tool, `llvm-to-smt`, is packaged in the `llvm-to-smt` directory. It contains the top-level script `generate_encodings.py`, which is essential for running our tool. The `llvm-passes` subdirectory includes our LLVM transformation passes and the required scripts for executing the passes. Our tool performs the following steps:

- Checks out the specified kernel version
- Edits the `verifier.c` file to add a stub function for each eBPF instruction that requires an SMT encoding.
- Extracts the compilation flags needed to convert the eBPF verifier kernel module to LLVM IR.
- Compiles `verifier.c` and `tnum.c` to `verifier.ll`.
- Applies the following LLVM passes from the `llvm-passes` subdirectory to transform the IR: `ForceFunctionEarlyExit`, `RemoveFunctionCalls`, `InlineFunctionCalls`, and `PromoteMemcpy`.
- Finally, runs the `LLVMToSMT` pass for each eBPF instruction to obtain the SMT encoding for it.

The script `generate_encodings.py` expects a git directory with the linux source code already downloaded (here, `/home/linux-stable/`). It is possible to obtain the SMT encodings for a different kernel version (greater than 4.14). It is also possible to obtain the SMT encoding for a single eBPF abstract operator using the `--specific-op` switch. For example, to get the encoding of `BPF_ADD` in kernel v5.10, we would need to perform the following steps:

```
mkdir /home/cav23-artifact/bpf-encodings-5.10
cd /home/cav23-artifact/llvm-to-smt
python3 generate_encodings.py --kernver 5.10 --kernbasedir /home/linux-stable --outdir /home/cav23-
artifact/bpf-encodings-5.10
```

Verification and synthesis

The next tool is packaged in `bpf-verification`. The script `bpf_alu_jmp_synthesis.py` script can be configured using `verification_synth_setup_config.toml` or command line arguments, which provies various options for verification and synthesis. In the previous experiment (2.1 and 2.2), we evaluated a specific kernel version for a specific set of eBPF instructions. The tool however, supports any kernel version after 4.14, and can accept any instruction(s) of interest. For example, if we want to test one instruction, `BPF_ADD`, in kernel version 5.10 we can use the following command:

```
python3 bpf_alu_jmp_synthesis.py --kernver 5.10 \
  --encodings_path <path to 5.10 encodings directory> \
  --ver_set BPF_ADD
```

If we want to test 2 instructions or more we add them sequentially as follows:

```
python3 bpf_alu_jmp_synthesis.py --kernver 5.10 \
  --encodings_path <path to 5.10 encodings directory> \
  --ver_set BPF_ADD BPF_OR BPF_AND
```

```
bpf-verification/
├── lib_reg_bounds_tracking
│   └── lib_reg_bounds_tracking.py
└── src
```

```
├── bpf_alu_jmp_synthesis.py
├── sync_soundness.py
├── synthesize_bug_types.py
├── verification_synth_setup_config.toml
└── wf_soundness.py
```

We use one script `bpf_alu_jmp_synthesis.py` to call three modules which perform verification and synthesis; two modules are used for performing verification, one for gen (`wf_soundness.py`) and one for sro(`sync_soundness.py`), and one more module for synthesis(`synthesize_bug_types.py`). Each of these modules uses the encodings produced by our llvm-to-smt procedure to verify instructions or synthesize POCs. Our verification conditions and concrete semantics for bpf instructions are contained within the `verification_synth_module` class included in the shared module library called `lib_reg_bounds_tracking.py` under the lib_reg_bounds_tracking directory.

Extensibility

Our paper and tool mainly focus on the range analysis in the eBPF verifier. However, it is possible to extend our tool to newer versions of the Linux kernel and to verify other analyses and potentially new abstract domains in the eBPF verifier.

To extend our work to other analyses, the tooling would first need to support the conversion of the C code related to those analyses to SMT. Our `llvm-to-smt` tool is modular and can support additional LLVM passes to handle new instruction sequences required for converting new parts of the verifier's C code to SMT.

After converting the C code to an LLVM form compatible with SMT translation, the LLVMtoSMT pass can be modified to handle a richer set of instructions. Currently, each LLVM instruction is handled in its own function within the `FunctionEncoder.cpp` class, but additional functions can be added to handle a wider range of LLVM instructions. More details about the tool are presented in §5 of the paper.

Once we obtain the semantics of the C code in first order, we can move on to verify the correctness of the other analyses done in the verifier. For this, we have to revise our verification conditions for future analyses. We specify these conditions in `lib_reg_bounds_tracking.py`. New domains can be supported by updating the existing verification conditions and relating the eBPF's concrete semantics with the verifier's abstract semantics.

Fin.