Summer NSF REU Activities Report

Josh Cooper

Supervising Professors: Santosh Nagarakatte, Srinivas Narayana

8/24/20

Project: eBPF Static Analysis and Translation into z3Py First Order Logic Formulas

1 Introduction

For the last three months, I have been attempting to familiarize myself with the workings of the eBPF instruction set used in the Linux kernel, and figure out how to implement an interpreter to transform that small set of instructions into a form that can be evaluated using the z3Py SMT Solver. This learning process has introduced me to a variety of unfamiliar aspects in computer science, both theoretical and executable, concerning how programs are created and run and how a compiler would work through the myriad ways a program could take a path from input to output. From learning a bit about lattice and graph theory, to implementing high performance graph algorithms using existing network library functions, this summer has been a fantastic learning opportunity for me, and has also produced the beginnings of a program which should help to further the overall goals of the project.

My personal contribution to the summers work has been developing a Python interpreter which can be given a specific eBPF program, and be asked to evaluate both the end result of the program, and whether there are any specific inputs which might cause the program to break. I have succeeded in the first task, and have developed a procedure to translate and work through a single path of a program given a specific subset of instructions and defined program inputs. Unfortunately, I was unable to complete the overall goal of transforming a full program into a function which can be evaluated over a possible range of inputs at once. There will be more discussion later on specific applications/drawbacks to my approach, and what future steps can be taken to expand my program.

2 Specific Program Goals

To build my eBPF to FOL interpreter, I had to accomplish two main goals. First, I would need to devise a method for translating the actions of a specific eBPF command into a FOL format, and second, I would need to string together a sequence of instructions in a manner which could be evaluated efficiently and correctly. To accomplish the first goal in the timeframe given to me for this project, I limited my investigations to a specific subset of instructions. The current instructions supported by my program are:

Supported Instructions	Supported Jump Statements
BPF_ADD (add two values together)	JMP_JNE (jump if not equal)
BPF_MOV (put a new value in a register)	JMP_EQ (jump if equal)
BPF_LSH (left shift a value)	JMP_GT (jump if greater than [unsigned])
BPF_RSH (logical right shift a value)	JMP_SGT (jump if greater than [signed])
BPF_ARSH (arithmetic right shift a value)	

These specific instructions were chosen for two reasons. They are the simplest to implement, each only requiring a single function that is already included with the z3Py library, and they do not require anything to be done with memory access for storing/retrieving information from an eBPF programs run. I will discuss the lack of memory access features in Section 4. In addition to the base instructions, all the above have support for taking values from registers already present in the program, and from outside sources, in either 32 bit or 64 bit inputs.

In order to link the individual instructions together in a manner that would allow for execution, I implemented a control flow graph made of up basic blocks from the main program containing some combination of the above instructions. Using that CFG, I execute a single pass through the program, taking a defined single path through from start to some end point, based on the specifics of the program and any predefined input values. The final output of the program will return the values held in any register referenced or changed in the program, or it will return an error message indicating the location and type of error that occurred in parsing through the program path.

3 Implementation Details

In order to explain how the program works, I'm going to run through the execution of a sample eBPF program, and describe the inner workings and eventual outputs. The specific program being run is the instructions3 test in inst_test.cc from the smartnic/superopt test cases, which is the source for all of the practice tests I am using as a general benchmark for program correctness in Section 4. In the following procedure, function names will be followed by a (1) or a (2), which will reference the specific file the function comes from. (1) refers to Basic_Block_CFG_Creator.py, and (2) refers to FOL_from_BPF.py. Instructions are in the form of {Keyword} {Destination} {Source}

Example Program:

	Instructions
0:	MOV32XC 0 -1
1:	ADD64XC 0 0x1
2:	MOV64XC 1 0x0
3:	JEQXC 0 0 4
4:	MOV64XC 0 -1
5:	JEQXC 0 0xffffffff 1
6:	EXIT
7:	MOV64XC 0 0
8:	EXIT

Step 1:

The instruction list is passed in to create_program(2), and a Program_Holder(2) object is created. Program_Holder objects call basic_block_CFG_and_phi_function_setup(1), which is a meta function for coordinating all other functions in Basic_Block_CFG_Creator.py.

Step 2:

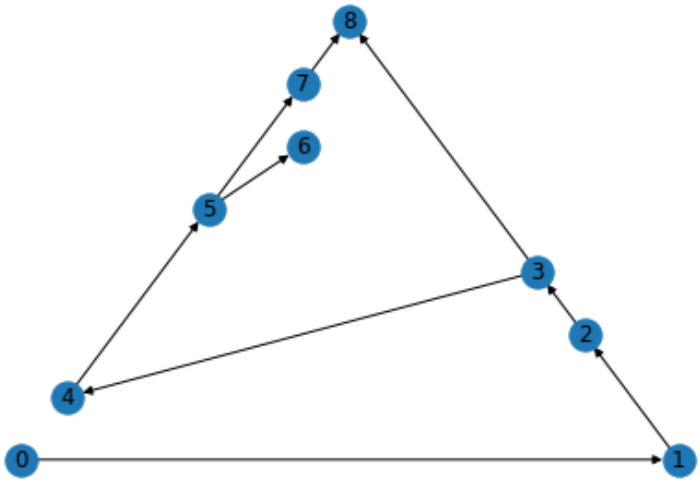
basic_block_CFG_and_phi_function_setup(1) calls two functions, set_up_basic_block_cfg(1) and phi_function_location(1). Phi_function_location is currently not used in the execution of the program, and is explained more in Section 5.

Step 3:

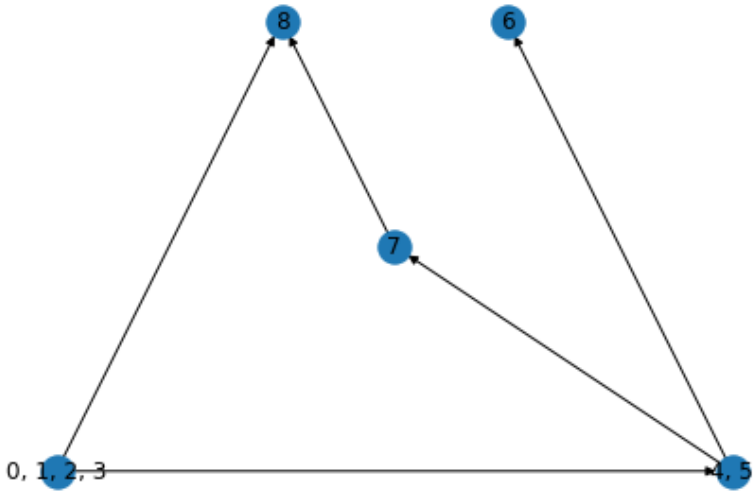
set_up_basic_block_cfg(1) handles the transformation of the passed in instruction list into the internal representation used for the evaluation of the instruction list. First, it transforms each individual instruction in the instruction list into an Instruction_Info(1) object, which acts as a container to hold specific information about that instruction. Instruction_Info objects hold the values used in the instruction, the connections from that specific instruction to other instructions, and the name of the register that will eventually hold the data. More on the choice of names for registers in a little while.

Step 4:

After creating all the individual Instruction_Info(1) objects, set_up_basic_block_cfg(1) creates a control flow graph for the individual instructions, and then separates the instruction list into basic blocks of straight-line code, creates a new Basic_Block(1) object, and links those blocks together in the CFG used for the main program execution. At this point in the set_up_basic_block_cfg(1) functions, there are also a pair of drawing options to visualize the graphs created for the instruction list and the basic blocks. For our example program, the instruction list CFG looks like (node names are the instruction number in the instruction list):



The basic block graph looks like this (node names are the instructions contained in the block):



After the basic block CFG has been created, the main functions in Basic_Block_CFG_Creator are ready to return to create_program and begin executing through the instruction list.

Step 5:

Now that we have created our CFG of Basic Blocks, we have to execute the program and find out what the output is. Create_program now calls the add_instruction_from_block(2) function, which starts to iterate through the CFG, executing the instructions stored in the first basic block.

Step 6:

Add_instruction_from_block(2) is where all the specific first order logic, register changes, and jump constraint evaluation occurs. Due to the use of the z3Py solver, any time we make a change to a register, we need to give it a new name so we don't lose the value by overwriting the old name in the z3Py Solver. We do this by using Static Single Assignment form for register names, where every time a register has a value changed in it, it is given a new name which references which instruction has specifically made the change. This fulfills both the requirements of the z3Py Solver, and allows us to print out the eventual conclusion of the z3Py Solver, and trace all changes made to all registers to identify any problem spots.

Based on the specific instructions residing in our current block, this function will either:

1. Create an FOL version of the instruction, and add it onto the formula created so far in the block
2. Evaluate an EXIT instruction, and return control of the program to create_program(2), which will cause the main function to end and print the results
3. Evaluate a jump instruction, which will decide which of the successor blocks will be moved to based on the satisfiability of the formula created by Step 1 and the specific jump condition

The jump condition will only every be checking for satisfiability based on the formula created inside its own specific block. The calculated register values will always be passed forward from a previous block, to limit the calculations needed to tell where the control flow needs to pass to next.

Step 7:

After a block has executed its instructions, and new values for the registers have been calculated, create_program(2) will continue to call add_instructions_from_block(2) until the program encounters an error, or finishes its execution. Now it will print out a summary of the results to the console, including the specific path of instructions taken, the overall runtime of the programs parts, and the ending results stored in all the registers. For our instruction list, the output is:

```
Adding instructions in block: 0, 1, 2, 3
    Checking Instruction: 0: MOV32XC 0 -1
    Checking Instruction: 1: ADD64XC 0 0x1
    Checking Instruction: 2: MOV64XC 1 0x0
    Checking Instruction: 3: JEQXC 0 0 4
Control moves to block: 8

Adding instructions in block: 8
    Checking Instruction: 8: EXIT

--> Program Results <--
    Model found the following results:
        Final Value for Register 0: 0
        Final Value for Register 1: 0
--> Total Run Time:         0.060 seconds <--
--> Time to make CFG:       0.018 seconds <--
--> Time to create FOL:     0.029 seconds <--
--> Time to Evaluate:       0.013 seconds <--
```

3.1 Note on Separation of Functions into Distinct Files:

The primary reason I have separated the two files used in this project is due to the possibilities for optimization on the two distinct aspects of my program. The graph creation portion is primarily based on using the networkX library, a highly optimized, very smart collection of insanely fast functions. The execution and evaluation of that graph was primarily written by myself, and will most likely have large numbers of optimizations possible in the future. By partitioning the creation of the graph away from the running of the

program, I hope to alleviate some of the confusion that might be present if I were to smash all the functions together in one file.

4 Results and Limitations

As of right now, I can pass 8 of the 9 tests using the smartnic instructions I have encoded. The only thing that is causing a problem is arithmetic right shift utilizing the lower 32 bits. Since the bitvectors I'm manipulating in the program are all 64 bits long, when the built in z3 rightshift executes, it fills in any open spots with the bit from the head of the 64 bit value, not the 32 bit value as needed. I've attempted a few fixes using masks and the z3 extract function, but nothing has worked yet from my changes. Otherwise, 32/64 bit Imm commands, and 32/64 bit register commands all work.

5 Future Program Extensions

One small area of growth for the program is adding in extra ALU or Jump instructions, which aren't included in the smartnic keyword list. This is not a difficult task, and would only involve adding an extra line or two for each new eBPF opcode type in the execute_instructions or check_jump function in FOL_from_BPF.py.

There are two large extensions that need to be made to this program. First, consideration for memory access and map use needs to be added in order to increase the scope and complexity of possible tested programs. Second, proper evaluation of the program as a whole, as opposed to single path evaluation, needs to be added, with the ability to query the formula for satisfiability over a range of inputs. The mechanisms for both these options eluded me this summer, but I think that the base program for CFG creation and instruction interpretation will allow for an easier time in the continuation of this project.

6 Personal Reflections

This summer research opportunity was the first time I have experienced a lot of the larger future aspects in computer science. Up to now, I had only ever worked on solo small side projects using knowledge I already possessed, or class projects which had very well defined criteria for success/failure. Having to jump into a project where no one knew what the exact form of our final output should be, and having to work hand in hand with the professors to flesh out the problem statement was a very rewarding experience. Working with a team, reporting on progress, and getting valuable real time feedback was also a first for me, and was very useful in my personal coding growth, specifically regarding my use of external libraries and existing theories. All told, this has been a fantastic growth experience, and I will most likely be utilizing skills gained from here for the rest of my career.

7 Source Code Documentation

All needed files in the gituhub folder located at:

https://github.com/JoshCoop1089/eBPF_Verification_Project/tree/master/Josh%20Code%20Tests/Code%20for%20Next%20Meeting

I also packaged everything into a zip just in case you think you forgot something. I included a writeup in READ_ME_FIRST.py on specific keywords and instruction formats, but the main gist of running a program is as follows.

1. Required Libraries
    a. networkX library -- https://networkx.github.io/documentation/stable/install.html
    b. pip install networkx
    c. z3Py library -- https://github.com/Z3Prover/z3
    d. pip install z3-solver
2. Runtime parameters (calling it parameters for this example)
    a. List of ints to start off the register values
3. eBPF Program (calling it instructions)
    a. list of strings in specific format
4. Import FOL_from_BPF
5. run create_program(instructions, parameters)
6. outputs register end results to console