Summer NSF REU Activities Report

Sammy Berger

Supervising Professors: Santosh Nagarakatte, Srinivas Narayana

8/24/20

Project: Verifying the Linux Kernel eBPF Verifier with Sa

# Introduction

The Linux Kernel is a large and daunting beast, with what seems like an arbitrarily large amount of code. For the past three months, I have been working alongside my research group to understand and analyze a small part of it – the eBPF verifier. This verifier attempts to automatically review eBPF programs and should only execute them if they subscribe to some intense safety conditions. Our goal was to create some level of proof of correctness of the verifier, or to instead find bugs. To do so, we used a powerful tool – a SAT solver. We specifically used the z3.py SMT Solver due to the easy interface and added utility that comes with using Python. Over the course of the summer, I’ve learned a lot about program verification on both a theoretical and practical level. On a theoretical level, I have progressed from what now seems basic – such as type verification algorithms – to lattice theory, which I still feel as though I’ve only brushed the surface of.

Over the course of the summer, I was also presented with the difficulties that come with taking on monumental tasks. Much of my work has been towards laying the groundwork for future work. I have explored multiple different ways to precisely analyze how a specific code snippet functions. My most immediately interesting work would be in my translation of the source code of one of the verifier’s C-based structs into a Python class designed to be manipulated and checked with a SAT solver. My code uses object-oriented ideas to tackle what might otherwise be complex SAT equations in a novel manner. There is further discussion as to the merits and demerits of this approach, as well as other approaches I considered and tested.

# Precise Problem Statement

The verifier will do several checks on any given instruction and on the entire eBPF program throughout the checking process. It then claims that if the program passes, it is ‘safe’ – that it follows certain safety conditions it lays out. Our original goal was to use a SAT solver to ask the following question: does there exist an input program *p* such that the verifier claims that every safety condition *s* holds for *p*, while there does exist some *s’* which *p* does violate?

Over the course of the summer, we identified that our end goal was to check two different satisfiability equations against each other. The ‘oracle safety condition’, which codifies what the verifier describes as safe, and the ‘verifier accept condition’, which codifies precisely what the verifier ensures. In these terms, our goal was simply to determine if there was an input which passed the ‘verifier accept condition’ and did not pass ‘oracle safety condition’.

# Approach

Josh worked on creating a Python-based verifier that could create oracle safety conditions for given code snippets. I worked on translating the verifier code from C to Python to utilize the z3.py SAT solver and create a verifier safety condition. In this report, I will only talk about my own contribution.

## Automated Representation of Execution

My first attempt was specifically with the goal in mind of automation. The main file for the verifier source code is already over ten thousand lines, and there are multiple other files which it accesses and uses which would also have to be examined rigorously. Therefore, my initial inclination was to manually mockup a translation which could be easily automated by a program. Below, you’ll see a side-by-side of the verifier source code, and the z3.py translation I produced.

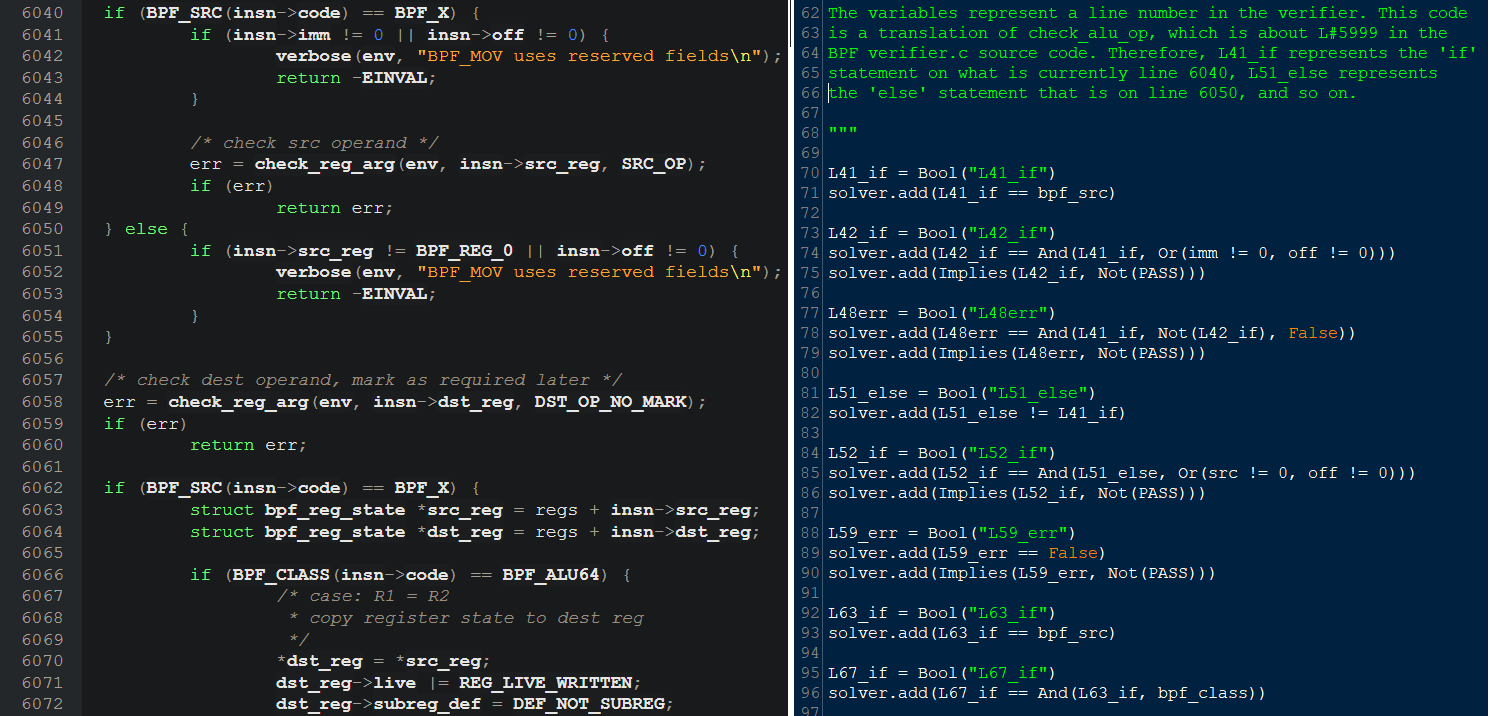


Figure : Verifier source code (left), Translation (right)

This code attempted to follow the execution path of the C source code. It works somewhat simplistically. It defines a Boolean SAT variable called PASS which it leaves undecided. At every branching path, it creates another Boolean variable, whose value is equal to the Boolean expression which determines if that path is taken. From here on in, I will refer to lines in the verifier source code with an L, followed by the line number (so the method check\_alu\_op would start at L5999). I will refer to lines in my code with the word Line, followed by the number (so this code snippet shows Lines 62-96). Hopefully this will reduce confusion.

As an example of how this code works, let’s look at L6041 and Lines 73-75. First, in L73 we declare a Boolean variable L42\_if, which represents whether L6041 executes in a runthrough of the verifier’s code. In Line 74, we assert to the SAT solver that L42\_if will only be true if two things hold true – the actual condition on L6041, and that the If statement on L6040 was true. Effectively, we also require that we reach this point in the code.

Then, we see that on L6043, the verifier will always return invalid if we pass L6041 successfully. Therefore, in Line 75, we assert that L42\_if implies Not(PASS). What does this ensure? Well, let us examine the truth table.

|  |  |  |
| --- | --- | --- |
| **L42\_if** | **Pass** | **Result** |
| True | True | False |
| True | False | True |
| False | True | True |
| False | False | True |

We can see that the SAT solver will fail to find a solution is if it somehow reaches a point where L42\_if holds, but it also requires that PASS be True. In other words, we require that when L6041 executes, Pass is False; and when L6041 does not execute, Pass can be either True or False. This is precisely the behavior we want.

In the rest of the program, whenever a path leads to an invalid return, we simply assert to the SAT solver that the previous line implies Not(PASS). The solver, at the time of completion, will have many asserts that say (bad\_path\_condition) implies Not(Pass). Then, when the program finishes, you can simply ask the solver to solve PASS == True. Every solution to that should be exactly every set of variables that the verifier will say passes.

## Direct Translation of Code

Eventually, though, I came to two hard walls. The first of which is the complexity of the verifier; a single BPF\_ADD instruction goes through a gamut of methods and adjust various bounds, update other state variables, and so on. This ties into the second issue, which is that the verifier uses structures such as the tnum and bpf\_verifier\_env, and they cannot be trivially reduced to more simple variables.

To solve both of these issues, I transitioned away from working with the high-level methods and started working with the more basic building blocks; specifically, the tnum struct source code.

# Results and Limitations

# Future Program Extensions

# Personal Reflections

# Source Code Documentation