Summer NSF REU Activities Report

Sammy Berger

Supervising Professors: Santosh Nagarakatte, Srinivas Narayana

Project: Verifying the Linux Kernel eBPF Verifier with SAT Solvers

1 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.

2 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'.

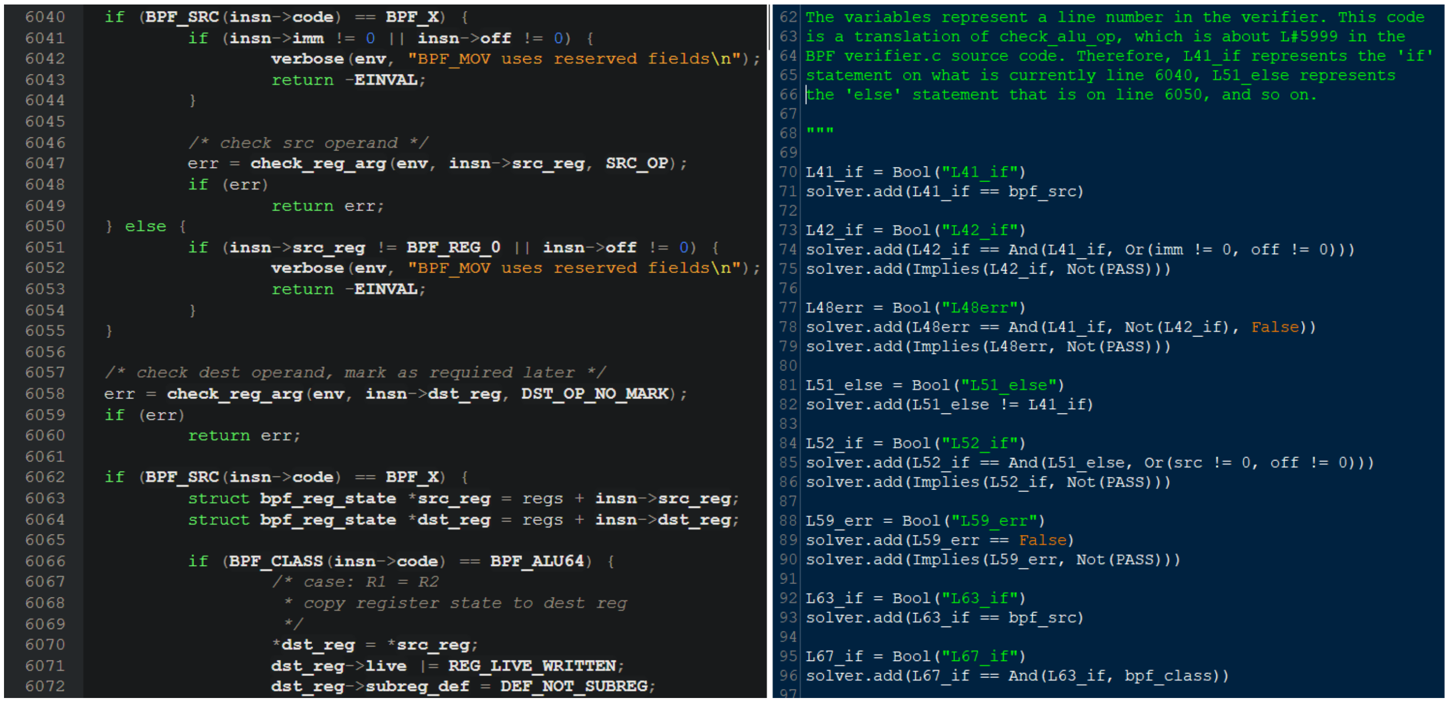
3 Approach

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 discuss my own contribution.

3.1 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.



```c
6040    if (BPF_SRC(insn->code) == BPF_X) {
6041            if (insn->imm != 0 || insn->off != 0) {
6042                    verbose(env, "BPF_MOV uses reserved fields\n");
6043                    return -EINVAL;
6044            }
6045
6046            /* check src operand */
6047            err = check_reg_arg(env, insn->src_reg, SRC_OP);
6048            if (err)
6049                    return err;
6050    } else {
6051            if (insn->src_reg != BPF_REG_0 || insn->off != 0) {
6052                    verbose(env, "BPF_MOV uses reserved fields\n");
6053                    return -EINVAL;
6054            }
6055    }
6056
6057    /* check dest operand, mark as required later */
6058    err = check_reg_arg(env, insn->dst_reg, DST_OP_NO_MARK);
6059    if (err)
6060            return err;
6061
6062    if (BPF_SRC(insn->code) == BPF_X) {
6063            struct bpf_reg_state *src_reg = regs + insn->src_reg;
6064            struct bpf_reg_state *dst_reg = regs + insn->dst_reg;
6065
6066            if (BPF_CLASS(insn->code) == BPF_ALU64) {
6067                    /* case: R1 = R2
6068                     * copy register state to dest reg
6069                     */
6070                    *dst_reg = *src_reg;
6071                    dst_reg->live |= REG_LIVE_WRITTEN;
6072                    dst_reg->subreg_def = DEF_NOT_SUBREG;
```

```python
62 The variables represent a line number in the verifier. This code
63 is a translation of check_alu_op, which is about L#5999 in the
64 BPF verifier.c source code. Therefore, L41_if represents the 'if'
65 statement on what is currently line 6040, L51_else represents
66 the 'else' statement that is on line 6050, and so on.
67
68 """
69
70 L41_if = Bool("L41_if")
71 solver.add(L41_if == bpf_src)
72
73 L42_if = Bool("L42_if")
74 solver.add(L42_if == And(L41_if, Or(imm != 0, off != 0)))
75 solver.add(Implies(L42_if, Not(PASS)))
76
77 L48err = Bool("L48err")
78 solver.add(L48err == And(L41_if, Not(L42_if), False))
79 solver.add(Implies(L48err, Not(PASS)))
80
81 L51_else = Bool("L51_else")
82 solver.add(L51_else != L41_if)
83
84 L52_if = Bool("L52_if")
85 solver.add(L52_if == And(L51_else, Or(src != 0, off != 0)))
86 solver.add(Implies(L52_if, Not(PASS)))
87
88 L59_err = Bool("L59_err")
89 solver.add(L59_err == False)
90 solver.add(Implies(L59_err, Not(PASS)))
91
92 L63_if = Bool("L63_if")
93 solver.add(L63_if == bpf_src)
94
95 L67_if = Bool("L67_if")
96 solver.add(L67_if == And(L63_if, bpf_class))
97
```

Figure 1: 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.

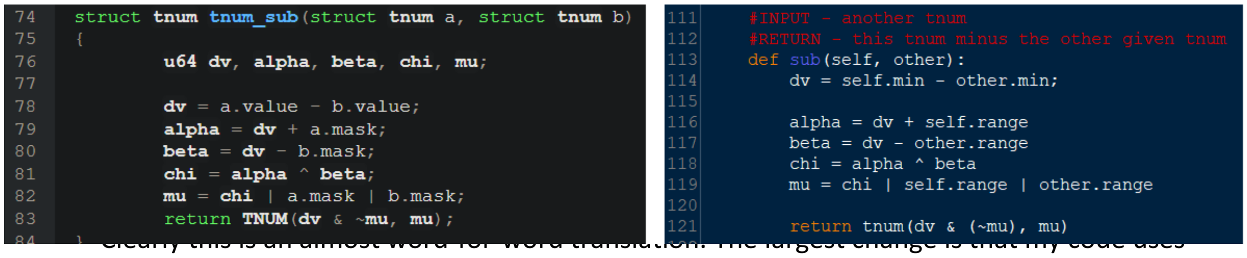
In the rest of the program, whenever a path leads to an invalid return, we simply assert to the SAT solver that the previous line implies Not(PASS). The solver, at the time of completion, will have many asserts that say (bad_path_condition) implies Not(Pass). Then, when the program finishes, you can 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.

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 to adjust various bounds it keeps track of, 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. Checking the direct execution of the program would require a working model or translation of those structures and might result in an unsolvably large SAT equation.

3.2 Direct Translation of Code

To solve both 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. I also radically changed my approach. I started to experiment with translating tnum into a Python class, with the goal of being able to use it in a SAT formula just as easily as a bit vector or an integer.

This approach proved to be fruitful. Below you'll see an excerpt of my translated tnum code side-by-side with the original c code.



```c
struct tnum tnum_sub(struct tnum a, struct tnum b)
{
        u64 dv, alpha, beta, chi, mu;

        dv = a.value - b.value;
        alpha = dv + a.mask;
        beta = dv - b.mask;
        chi = alpha ^ beta;
        mu = chi | a.mask | b.mask;
        return TNUM(dv & ~mu, mu);
```

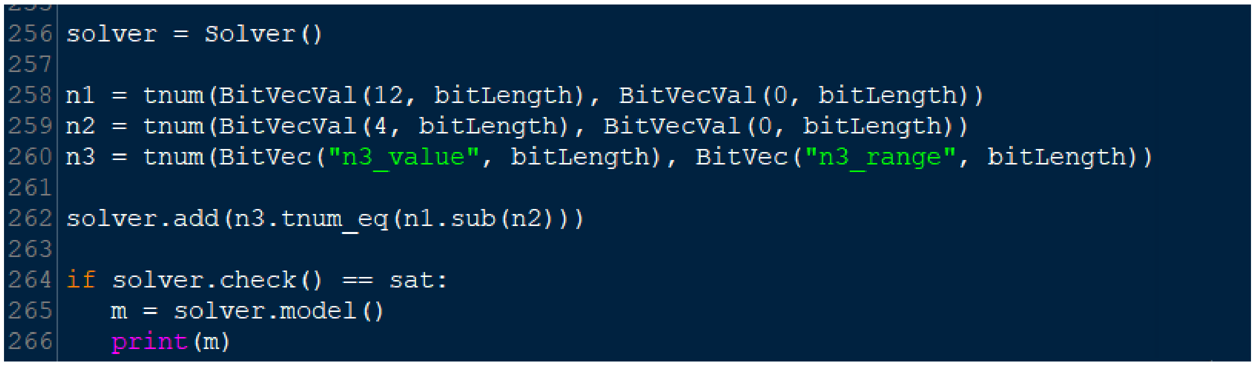
```python
    #INPUT - another tnum
    #RETURN - this tnum minus the other given tnum
    def sub(self, other):
        dv = self.min - other.min;

        alpha = dv + self.range
        beta = dv - other.range
        chi = alpha ^ beta
        mu = chi | self.range | other.range

        return tnum(dv & (~mu), mu)
```

Figure 2: Verifier source code (left), Translation (right)

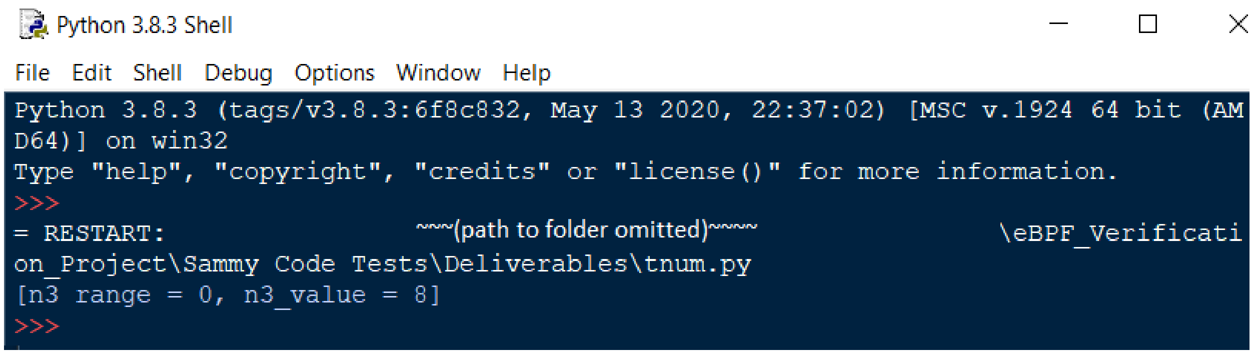
Clearly this is an almost word for word translation. The largest change is that my code uses

it easier to understand what the tnum represents.

The test below creates three tnums; n1, n2, and n3. It defines two tnums using z3.py bit vectors. BitVecVals are constant values, whereas BitVecs are variable bit vectors that the SAT solver will set. For the sake of this example, I kept the values easy to understand – n1 has a min/value of 12, which is 1100 in binary. The other discrete tnum n2 has a min/value of 4, which is 0100 in binary. I set both masks to 0 for simplicity's sake and asked the solver to satisfy that the unspecified n3 is equal to n2 subtracted from n1.



```
solver = Solver()

n1 = tnum(BitVecVal(12, bitLength), BitVecVal(0, bitLength))
n2 = tnum(BitVecVal(4, bitLength), BitVecVal(0, bitLength))
n3 = tnum(BitVec("n3_value", bitLength), BitVec("n3_range", bitLength))

solver.add(n3.tnum_eq(n1.sub(n2)))

if solver.check() == sat:
    m = solver.model()
    print(m)
```



```
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:37:02) [MSC v.1924 64 bit (AM
D64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART:                    ~~~(path to folder omitted)~~~~               \eBPF_Verificati
on_Project\Sammy Code Tests\Deliverables\tnum.py
[n3_range = 0, n3_value = 8]
>>>
```

Figure 3: Checking an operation on concrete values

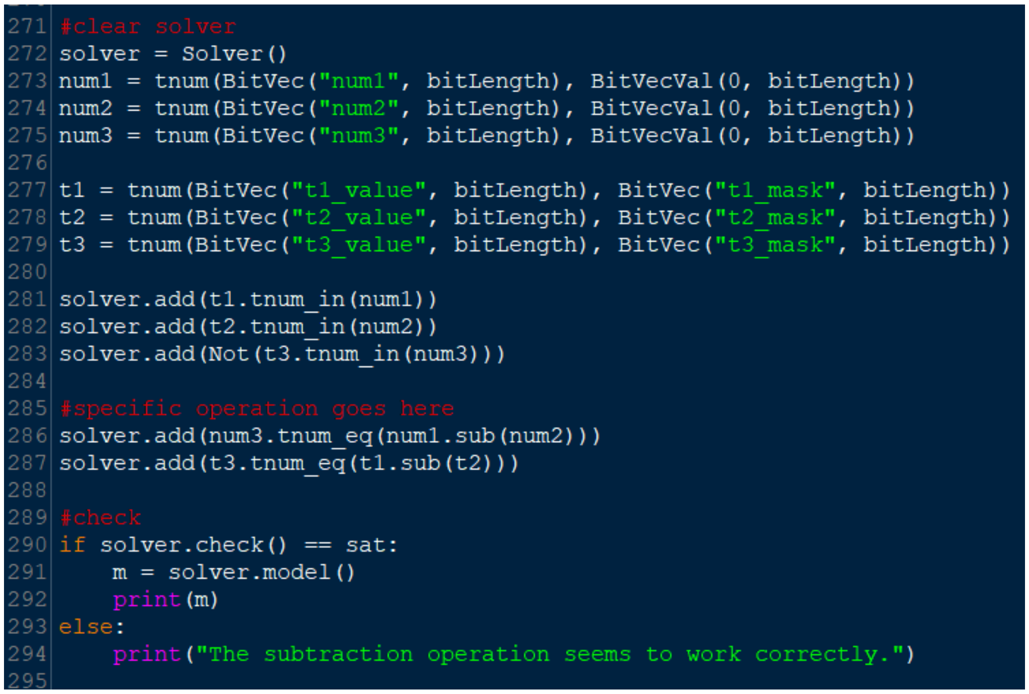
As you can see, the solver correctly identified that the result BitVector would have a value of 8 and a range of 0.

4 Results and Limitations

I did get limited results with the Representation of Execution approach, but as previously discussed I came across some seemingly insurmountable issues. SAT solvers are not perfect and can have issues if the number of restrictions grows too large. My limited scope translation in this manner functioned well and returned solutions quickly – in under a second, generally.

However, for this approach to yield any results would require that the SAT formula remains quickly solvable, and that is not necessarily guaranteed.

The Direct Translation approach currently feels more interesting to me. As of now, I have used it to prove that addition and subtraction of tnums works as it should. Below is the test case that checks whether subtraction is functional.



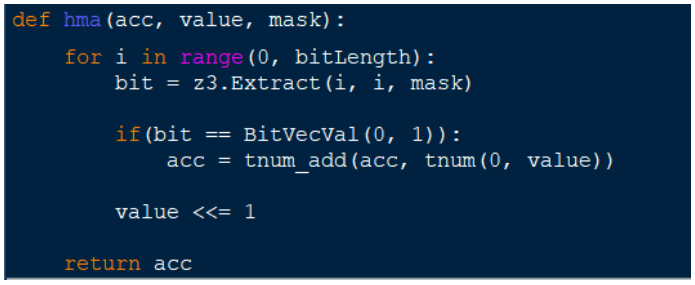
```
#clear solver
solver = Solver()
num1 = tnum(BitVec("num1", bitLength), BitVecVal(0, bitLength))
num2 = tnum(BitVec("num2", bitLength), BitVecVal(0, bitLength))
num3 = tnum(BitVec("num3", bitLength), BitVecVal(0, bitLength))

t1 = tnum(BitVec("t1_value", bitLength), BitVec("t1_mask", bitLength))
t2 = tnum(BitVec("t2_value", bitLength), BitVec("t2_mask", bitLength))
t3 = tnum(BitVec("t3_value", bitLength), BitVec("t3_mask", bitLength))

solver.add(t1.tnum_in(num1))
solver.add(t2.tnum_in(num2))
solver.add(Not(t3.tnum_in(num3)))

#specific operation goes here
solver.add(num3.tnum_eq(num1.sub(num2)))
solver.add(t3.tnum_eq(t1.sub(t2)))

#check
if solver.check() == sat:
    m = solver.model()
    print(m)
else:
    print("The subtraction operation seems to work correctly.")
```

Figure 4: Checking if an operation is valid over all inputs

When run, this code outputs that subtraction works correctly. Addition also functions properly. However, there are other issues to investigate.

A principal cause for concern is that z3.py uses bit vectors that are by default interpreted as signed; the actual tnum.c uses exclusively unsigned bit vectors. This can lead to issues, especially with the rightshift and leftshift operations.

Multiplication sometimes works and sometimes doesn't. Small numbers without masks that don't approach overflow work fine – the verifier correctly outputs that 2*4 = 8 when using the approach in Figure 3. However, the checking process in Figure 4 fails on multiplication. I suspect that this is not a flaw inherent in the algorithm or the C source code; rather, I believe it is in my translation, and is probably caused by faulty shifts in the hma() method, shown below.



```
def hma(acc, value, mask):

    for i in range(0, bitLength):
        bit = z3.Extract(i, i, mask)

        if(bit == BitVecVal(0, 1)):
            acc = tnum_add(acc, tnum(0, value))

        value <<= 1

    return acc
```

Figure 5: hma translation

5 Future Program Extensions

It is quite easy to see how to continue working towards representing the verifier's execution – the first step would be to automate the translation process. There are still questions to be answered, especially in terms of representing the verifier state and calling a method multiple times. However, I believe these to be surmountable. The real test would be if the resultant SAT formula can be processed by a solver.

The Direct Translation interests me because it touches on interesting ideas. By using the power of Python and z3.py you can create unspecified classes and objects which can later by checked by the solver. This allows you to pass tnum objects to other methods and use them in interesting ways, and only do final checks at the end. I think this could be used to great effect, especially when translating large code bases into a satisfiable form.

6 Personal Reflections

This summer felt a bit like I intended to launch a rocket; but instead, I was only able to hike through a forest to the proper location and set up the launch pad.

To be clear, I think the first four to six weeks of this experience were very valuable. I learned a lot about common program verification techniques, and past examples of vulnerabilities which were found in large important codebases like Mozilla Firefox. That knowledge was important in guiding me through the rest of the work I did during the summer, and I doubt I would have been as efficient if I hadn't spent that time learning, testing approaches, and trying new things.

Often, the hardest part of solving a problem is correctly identifying the question to ask and identifying the path to the solution. I think that during this summer I have at least done both of those things, even if I didn't finish the solution like I wanted to.

7 Source Code Documentation

The code for this project can be found in my partner Josh's github at:

https://github.com/JoshCoop1089/eBPF_Verification_Project/

My work is primarily in eBPF_Verification_Project/Sammy Code Tests, and most of the work that's fit to read is in eBPF_Verification_Project/Sammy Code Tests/Deliverables. In this report, I've mostly referenced two files: check_alu_op_translation.py, which is the automated-style translation of program execution, and tnum.py, which is the direct translation of the tnum.c code from the verifier.

I have taken the time to ensure that both files are well-commented. Hopefully, they are understandable. If you have any questions, you can contact me at sammy27berger@gmail.com.