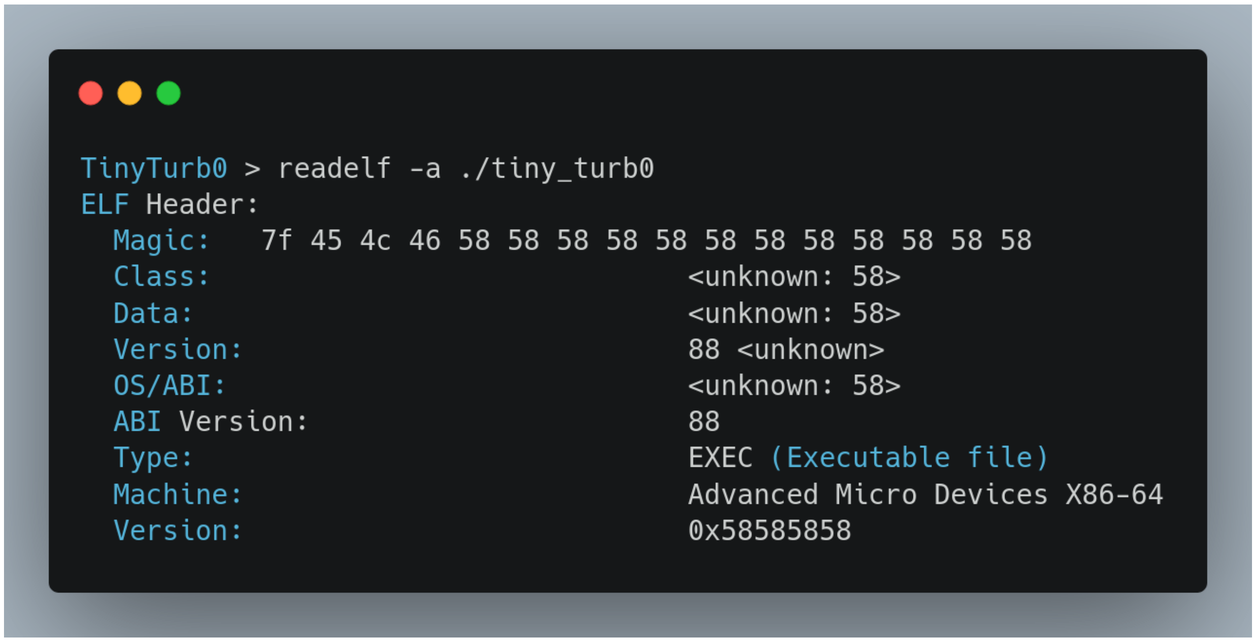
TinyTurb0

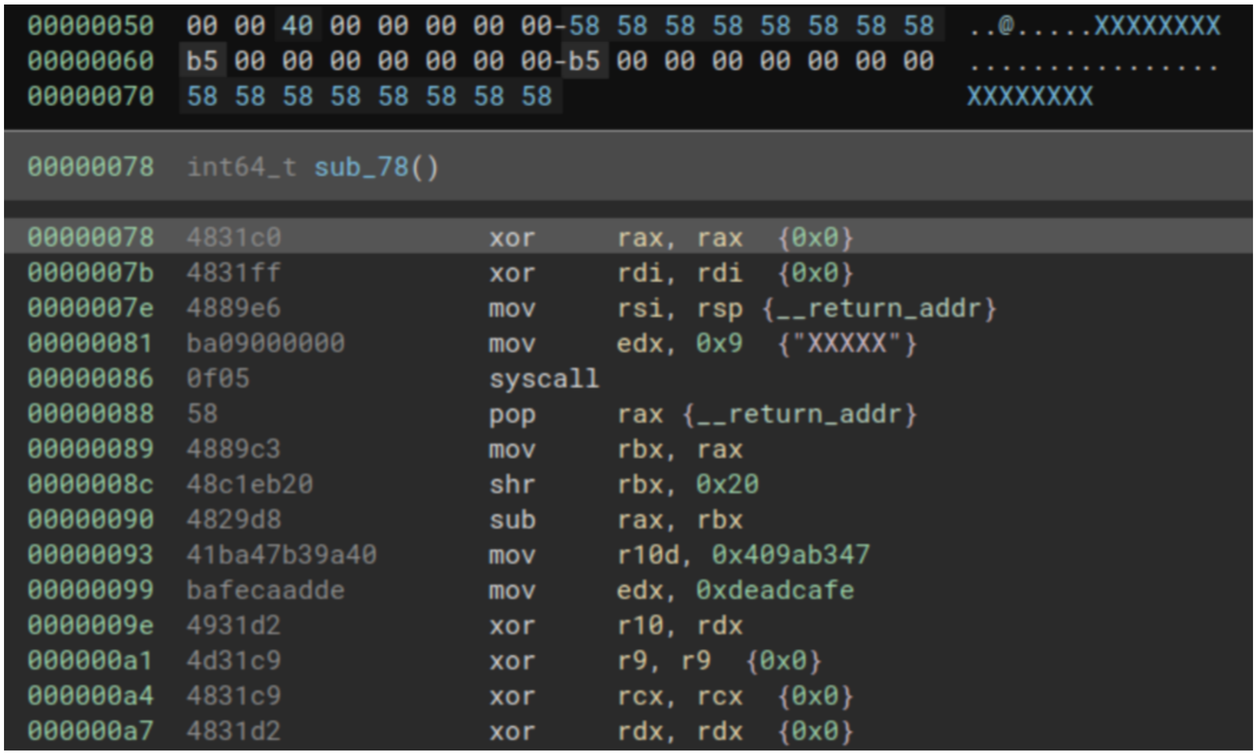
The Problem

We're given a tiny x86_64 binary (only 301 bytes!), which uses TheXCellerator's libgolf to mangle the ELF header. The binary is runnable on 64-bit Linux, but breaks many static analysis tools that trust the ELF header.



```
TinyTurb0 > readelf -a ./tiny_turb0
ELF Header:
  Magic:   7f 45 4c 46 58 58 58 58 58 58 58 58 58 58 58 58
  Class:                             <unknown: 58>
  Data:                              <unknown: 58>
  Version:                           88 <unknown>
  OS/ABI:                            <unknown: 58>
  ABI Version:                       88
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x58585858
```

BinaryNinja also has trouble, but by using "Make Function at This Address" at byte 0x78 (the end of the ELF and Program header), we can read the disassembly.



```
00000050  00 00 40 00 00 00 00 00-58 58 58 58 58 58 58 58  ..@.....XXXXXXXX
00000060  b5 00 00 00 00 00 00 00-b5 00 00 00 00 00 00 00  ................
00000070  58 58 58 58 58 58 58 58                          XXXXXXXX

00000078  int64_t sub_78()

00000078  4831c0          xor     rax, rax  {0x0}
0000007b  4831ff          xor     rdi, rdi  {0x0}
0000007e  4889e6          mov     rsi, rsp {__return_addr}
00000081  ba09000000      mov     edx, 0x9  {"XXXXX"}
00000086  0f05            syscall
00000088  58              pop     rax {__return_addr}
00000089  4889c3          mov     rbx, rax
0000008c  48c1eb20        shr     rbx, 0x20
00000090  4829d8          sub     rax, rbx
00000093  41ba47b39a40    mov     r10d, 0x409ab347
00000099  bafecaadde      mov     edx, 0xdeadcafe
0000009e  4931d2          xor     r10, rdx
000000a1  4d31c9          xor     r9, r9  {0x0}
000000a4  4831c9          xor     rcx, rcx  {0x0}
000000a7  4831d2          xor     rdx, rdx  {0x0}
```

The shellcode is quite simple. It reads 8 bytes from stdin, does various operations to these bytes in a 32 iteration loop, then compares the output to a fixed value (0xa7dd46516fefb265). If equal, it prints "1", otherwise nothing. The obvious goal is to find the correct input to satisfy this condition.

The hard part is the "various operations" in the loop. The length and complexity of the operations in the loop are simple enough to understand going forward, but a cursory attempt to reverse the instructions proved impossible.

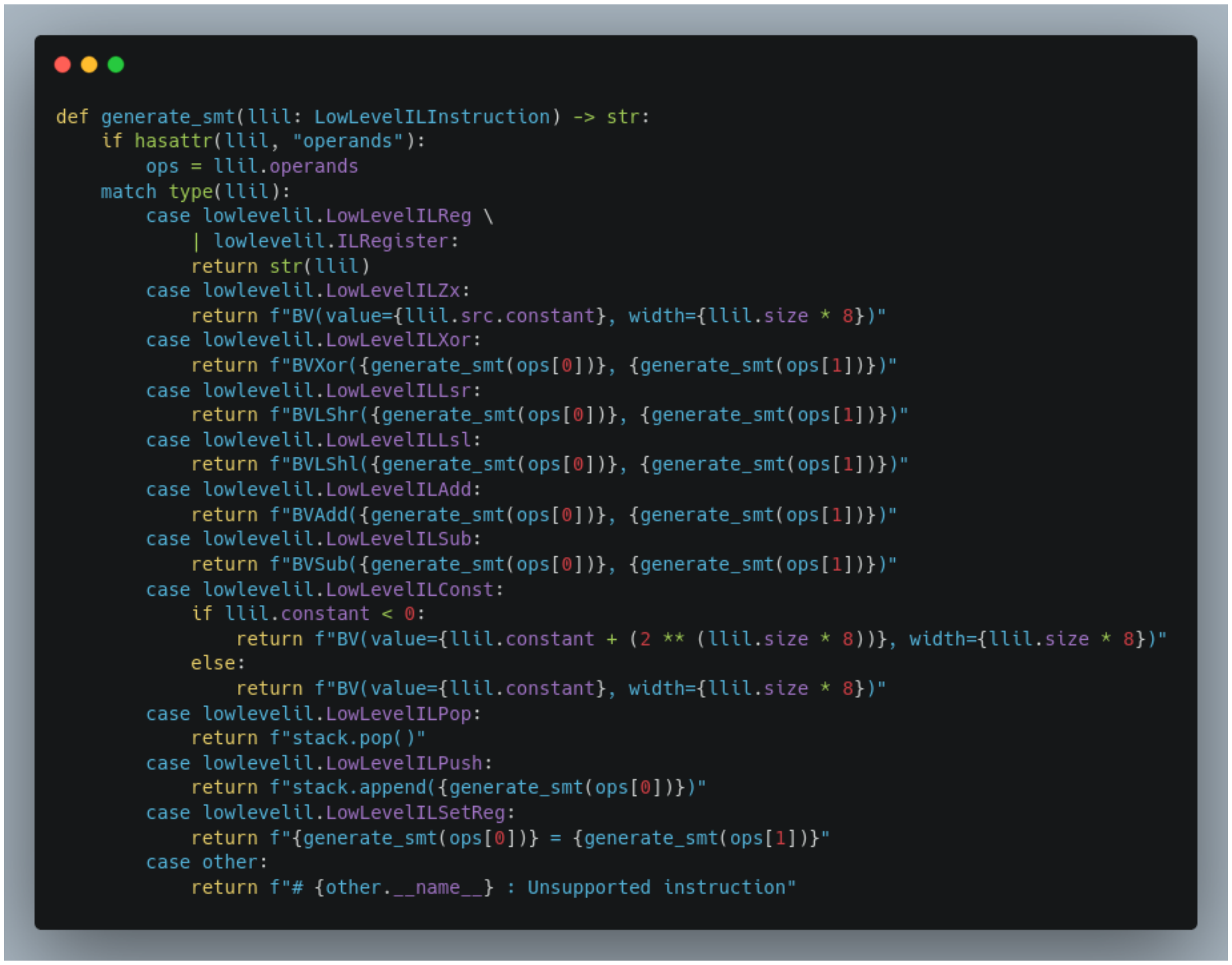
The Solution

To solve, we used pySMT, an API in python that can encode an Satisfiable Modulo Theory (SMT) problem in a single format and use multiple underlying solvers on it such as z3 or SMT-SAT.

The form of our pySMT solution is to symbolize the initial input, encode the operations done on that symbol, set it equal to the fixed value, and finally run it through a solver to find the initial value.

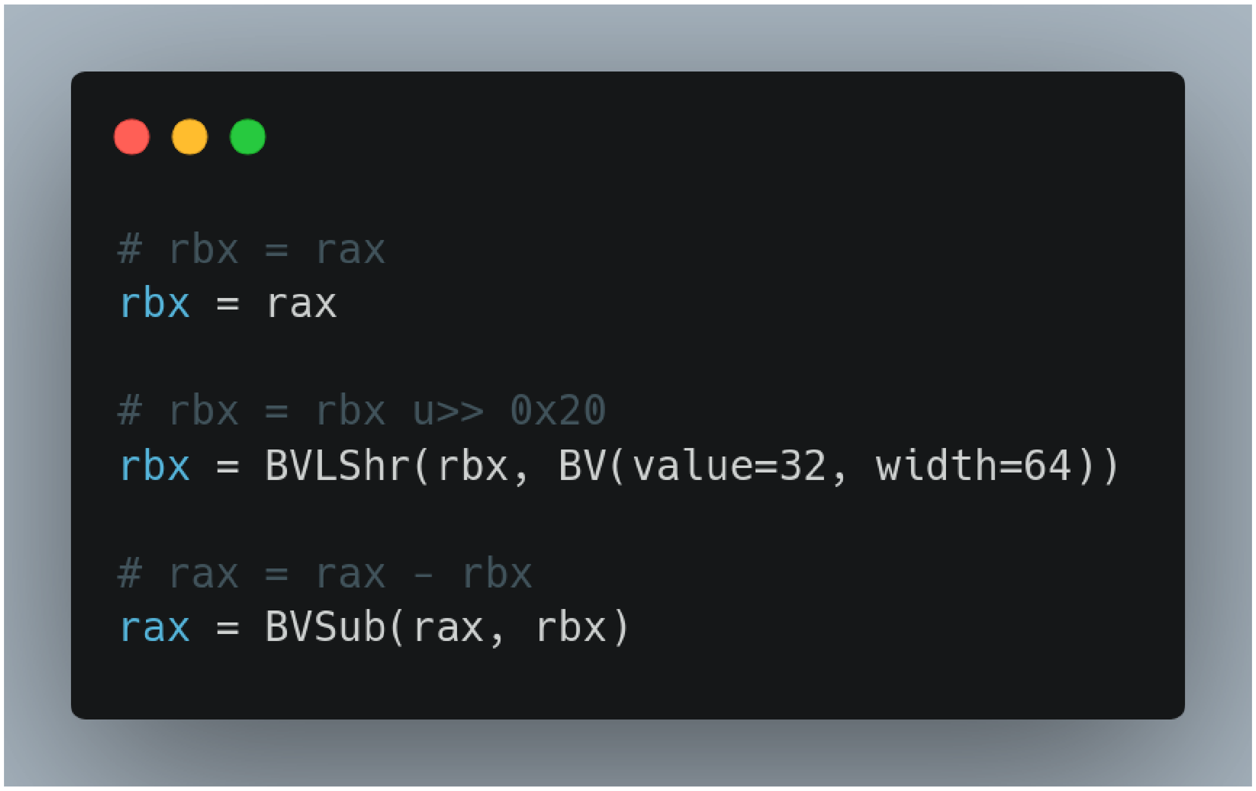
Thankfully pySMT includes a BitVector (`BV`) class where 32 or 64 bit registers can by symbolized, encoded, and operated on using the equivalent x86_64 instructions. For example, the x86 instruction to shift bits to the right, `shr`, can be translated into `BVLShr`. Encoding the registers as an Integer class in pySMT would not be sufficient as operations that cause overflows or 32-64 bit translations would not work.

Instead of translating the instructions into pySMT code manually, we wrote a code generator using the BinaryNinja python API. The [Low Level Instruction Language with its tree-like instructions paired well with modern python to make nice recursive function to generate the pySMT code:



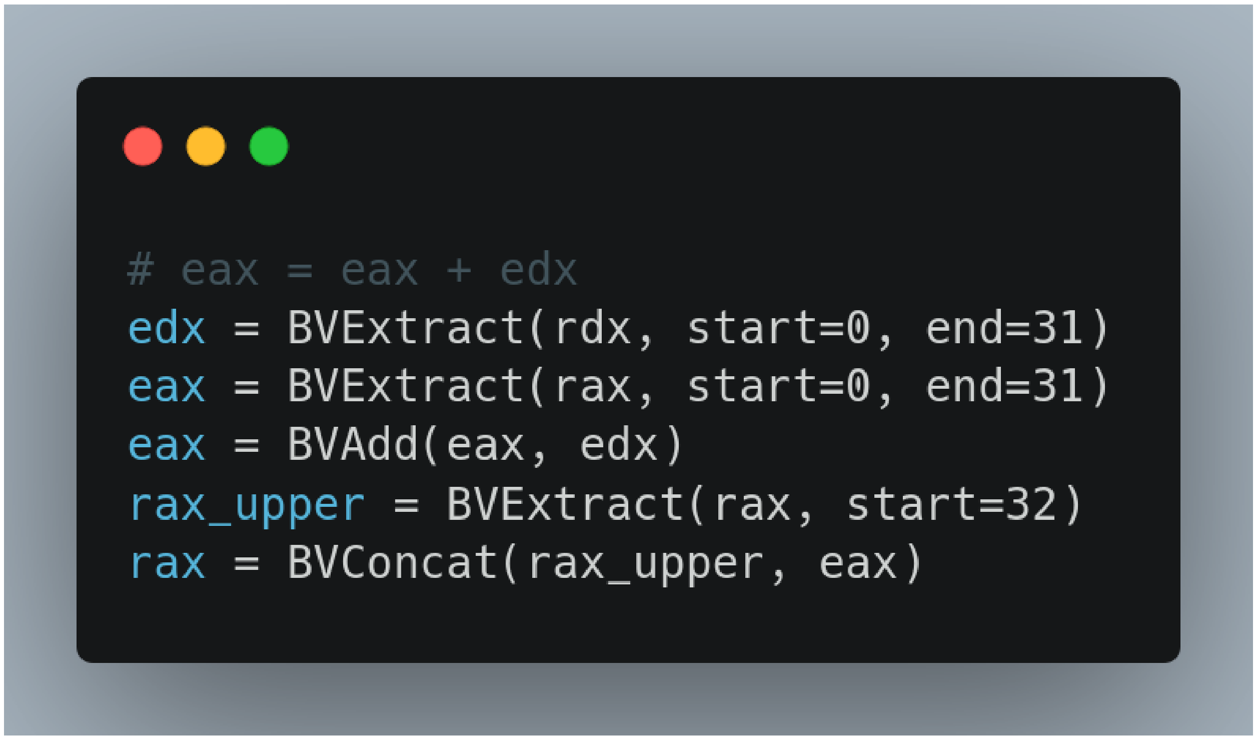
```python
def generate_smt(llil: LowLevelILInstruction) -> str:
    if hasattr(llil, "operands"):
        ops = llil.operands
    match type(llil):
        case lowlevelil.LowLevelILReg \
            | lowlevelil.ILRegister:
            return str(llil)
        case lowlevelil.LowLevelILZx:
            return f"BV(value={llil.src.constant}, width={llil.size * 8})"
        case lowlevelil.LowLevelILXor:
            return f"BVXor({generate_smt(ops[0])}, {generate_smt(ops[1])})"
        case lowlevelil.LowLevelILLsr:
            return f"BVLShr({generate_smt(ops[0])}, {generate_smt(ops[1])})"
        case lowlevelil.LowLevelILLsl:
            return f"BVLShl({generate_smt(ops[0])}, {generate_smt(ops[1])})"
        case lowlevelil.LowLevelILAdd:
            return f"BVAdd({generate_smt(ops[0])}, {generate_smt(ops[1])})"
        case lowlevelil.LowLevelILSub:
            return f"BVSub({generate_smt(ops[0])}, {generate_smt(ops[1])})"
        case lowlevelil.LowLevelILConst:
            if llil.constant < 0:
                return f"BV(value={llil.constant + (2 ** (llil.size * 8))}, width={llil.size * 8})"
            else:
                return f"BV(value={llil.constant}, width={llil.size * 8})"
        case lowlevelil.LowLevelILPop:
            return f"stack.pop()"
        case lowlevelil.LowLevelILPush:
            return f"stack.append({generate_smt(ops[0])})"
        case lowlevelil.LowLevelILSetReg:
            return f"{generate_smt(ops[0])} = {generate_smt(ops[1])}"
        case other:
            return f"# {other.__name__} : Unsupported instruction"
```

Generates code that looks like this:



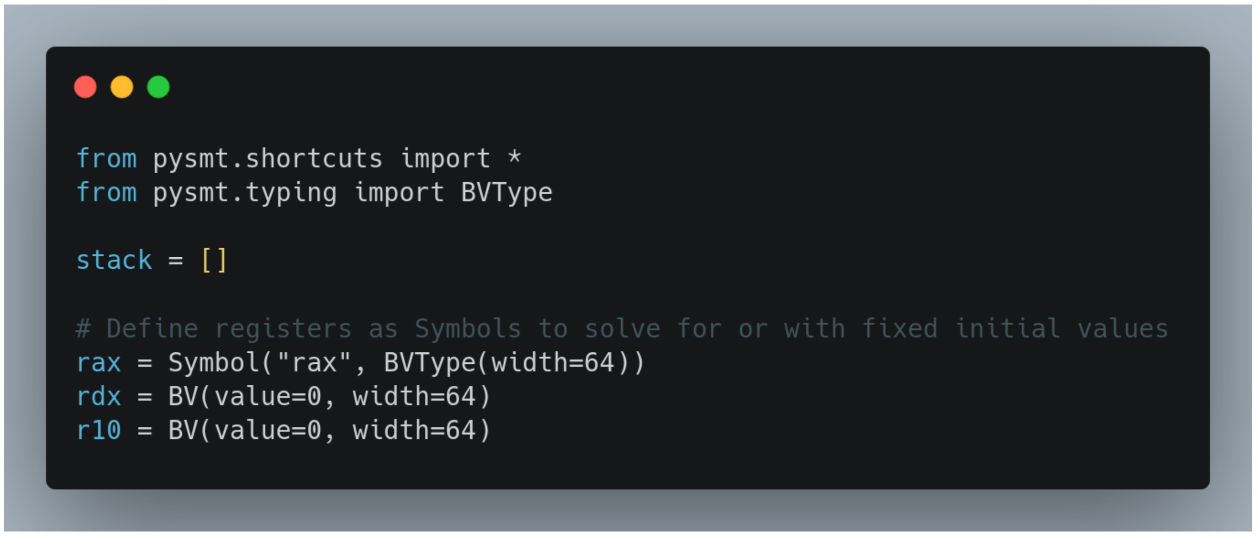
```
# rbx = rax
rbx = rax

# rbx = rbx u>> 0x20
rbx = BVLShr(rbx, BV(value=32, width=64))

# rax = rax - rbx
rax = BVSub(rax, rbx)
```

We also want to automatically translate between 32 and 64 bit register values, which we can do with pySMT's useful `BVExtract` and `BVConcat` functions:



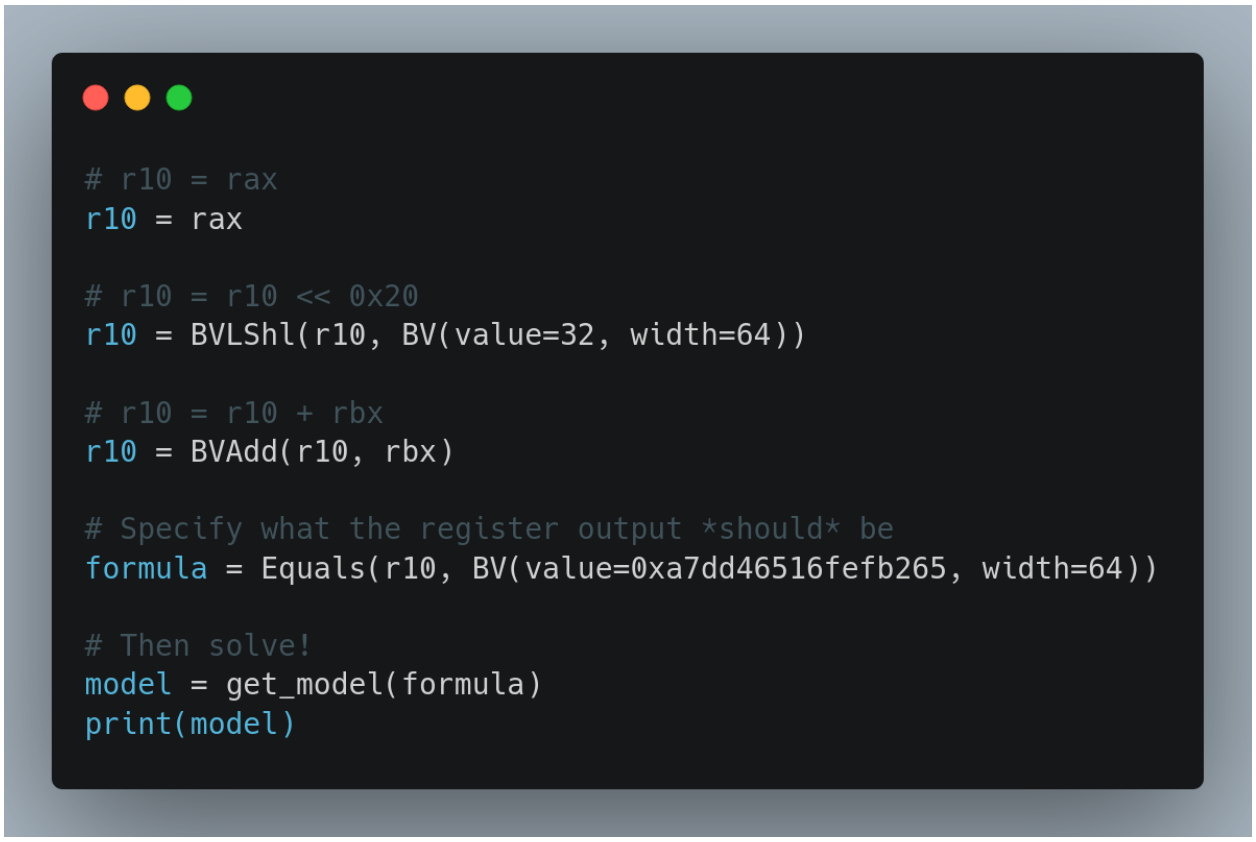
```
# eax = eax + edx
edx = BVExtract(rdx, start=0, end=31)
eax = BVExtract(rax, start=0, end=31)
eax = BVAdd(eax, edx)
rax_upper = BVExtract(rax, start=32)
rax = BVConcat(rax_upper, eax)
```

After generating the code from LLIL, we prepend the generated code with the correct Symbols and initial states:



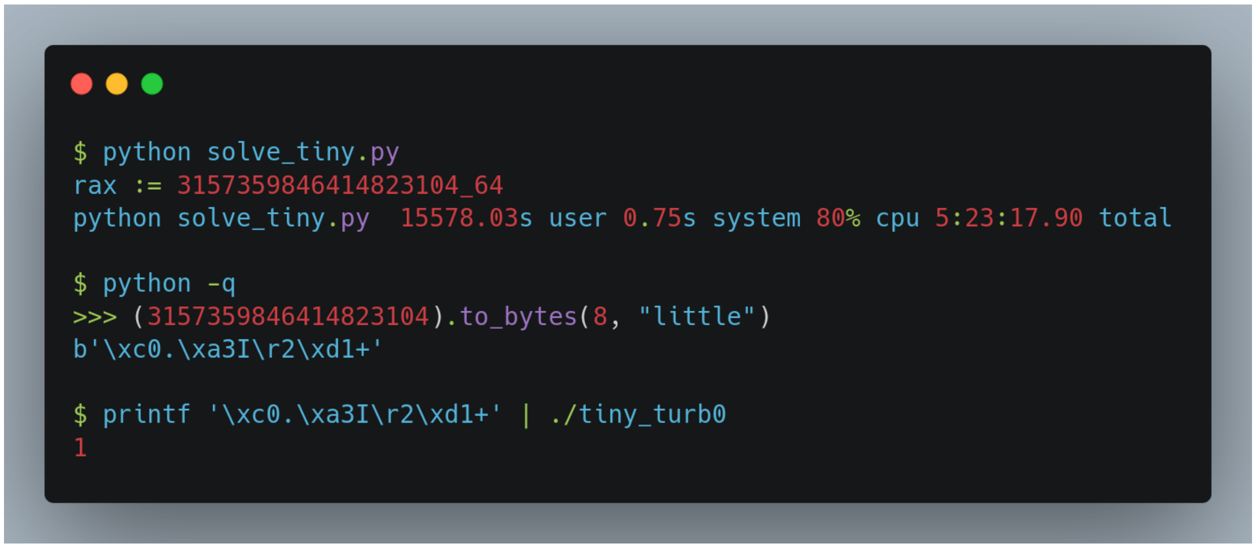
```python
from pysmt.shortcuts import *
from pysmt.typing import BVType

stack = []

# Define registers as Symbols to solve for or with fixed initial values
rax = Symbol("rax", BVType(width=64))
rdx = BV(value=0, width=64)
r10 = BV(value=0, width=64)
```

add the 32 iteration loop manually (If and Goto instructions are not supported in code gen), and append the code to actually solve for the expected value:



```python
# r10 = rax
r10 = rax

# r10 = r10 << 0x20
r10 = BVLShl(r10, BV(value=32, width=64))

# r10 = r10 + rbx
r10 = BVAdd(r10, rbx)

# Specify what the register output *should* be
formula = Equals(r10, BV(value=0xa7dd46516fefb265, width=64))

# Then solve!
model = get_model(formula)
print(model)
```

We can finally run the (slightly edited) generated code to solve for the correct value. This took about 4.5 hours on my machine:



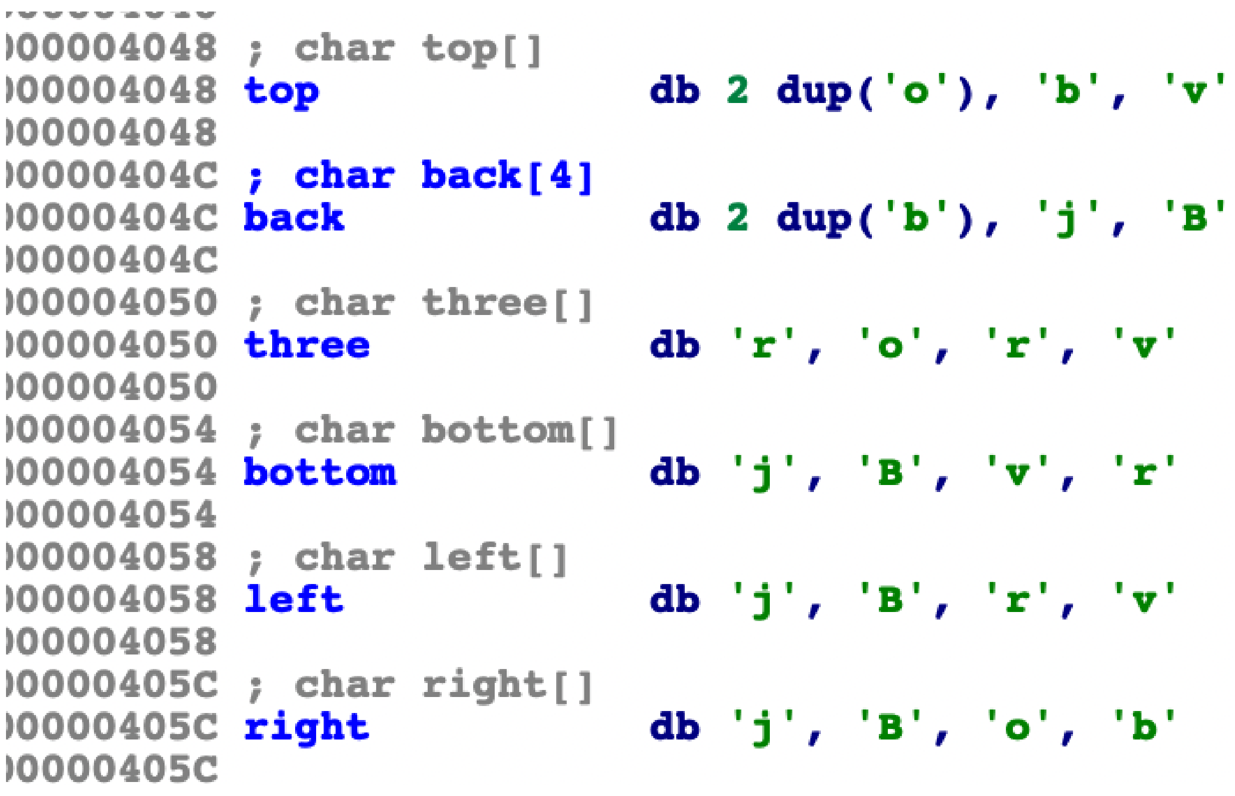
```
$ python solve_tiny.py
rax := 3157359846414823104_64
python solve_tiny.py  15578.03s user 0.75s system 80% cpu 5:23:17.90 total

$ python -q
>>> (3157359846414823104).to_bytes(8, "little")
b'\xc0.\xa3I\r2\xd1+'

$ printf '\xc0.\xa3I\r2\xd1+' | ./tiny_turb0
1
```

Full code for this challenge is found in TinyTurb0.tar.gz

StrangeCube

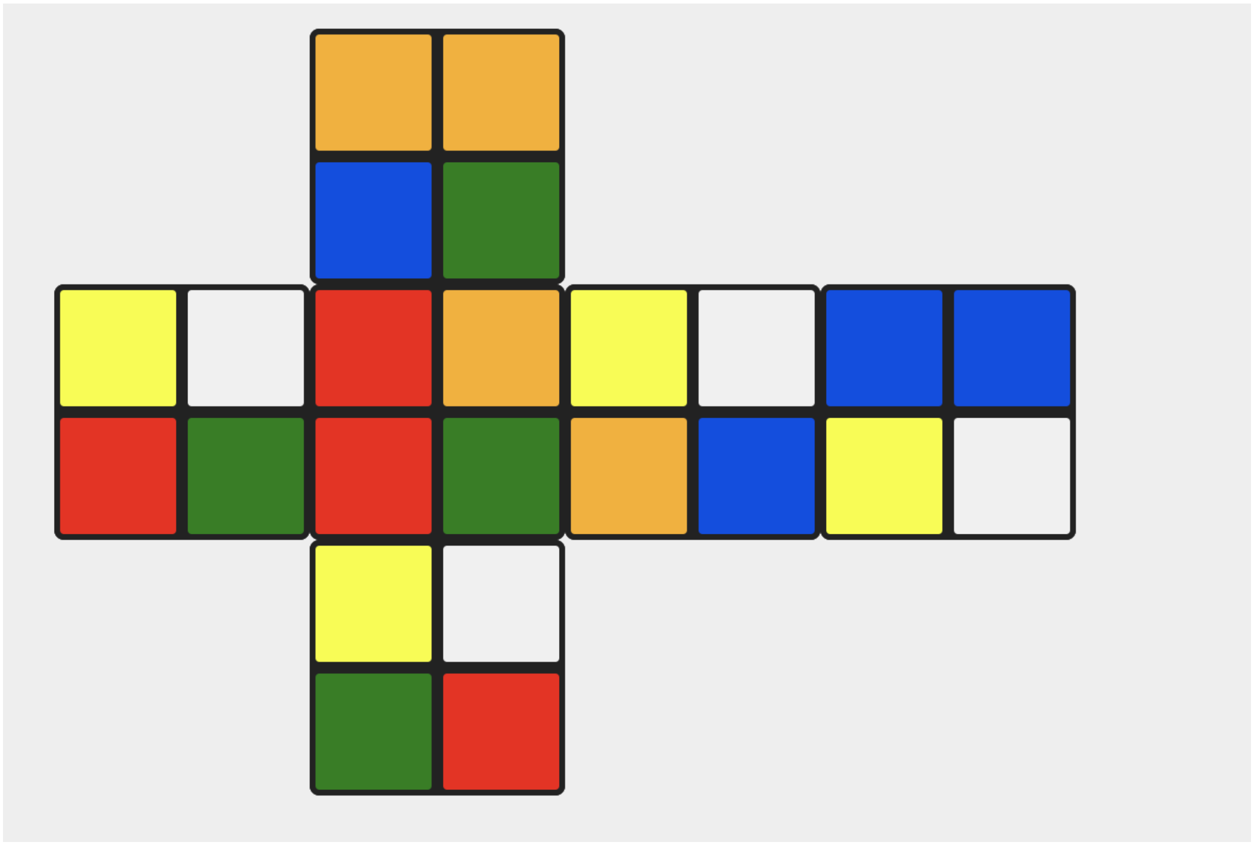
This reversing challenge was a 2x2x2 rubik's cube. First we mapped the commands to the related rubik’s moves that we found here: https://jperm.net/3x3/moves , once this was done we had to determine what the starting state of the rubik’s cube was.

The rubik’s cube faces were organized in memory as seen below, the labels were set according to the standard rubik’s cube face names (three == front).



```
)00004048 ; char top[]
)00004048 top             db 2 dup('o'), 'b', 'v'
)00004048
)0000404C ; char back[4]
)0000404C back            db 2 dup('b'), 'j', 'B'
)0000404C
)00004050 ; char three[]
)00004050 three           db 'r', 'o', 'r', 'v'
)00004050
)00004054 ; char bottom[]
)00004054 bottom          db 'j', 'B', 'v', 'r'
)00004054
)00004058 ; char left[]
)00004058 left            db 'j', 'B', 'r', 'v'
)00004058
)0000405C ; char right[]
)0000405C right           db 'j', 'B', 'o', 'b'
)0000405C
```

Plugging these values into a rubik's cube solver found online, we can see that it visually looks like this and get a solution!



```
~# ./strange_cube URRfRUfu
[*] G00d flag!
```

QuantumMatrix

Discovery

The binary given for this task is a stripped ELF file. It has to be run on a x86_64 Linux platform.

```
$ file ./qm
qm: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=e3b4dbda07a2220e909303535e23134e1a9b073c, for GNU/Linux 4.4.0,
stripped
```

The binary is relatively small, and has a small number of functions. Most of the code is in the `main` function. Below is a list of the program-specific functions. The other functions are added by the compiler and can be safely ignored.

Function	Size
fun_11d9	171
fun_1284	176
fun_1334	327
fun_147b	167
fun_1522	3592

Small functions

The function at 0x147b calls sqrt on the sum of the squares of its first argument. This is the [Euclidean norm] of a vector.

```
double norm(double *data, int size)
{
  double sum = 0;

  for(int i = 0; i < size; i++)
    sum += data[i] * data[i];

  return sqrt(sum);
}
```

[Euclidean norm] https://en.wikipedia.org/wiki/Vector_norm#Euclidean_norm

The functions at 0x11d9, 0x1284 and 0x1334 appear to work on a strange structure that contains two floats.

```
struct duo {
  float left;
  float right;
};
```

The function at 0x1334 does the following operation:

```
struct duo fun_1334(struct duo *a, struct duo *b, int n)
{
  struct duo r = {0, 0};

  for(int i = 0; i < n; i++) {
    r.left  += a[i].left * b[i].left  - a[i].right * b[i].right;
    r.right += a[i].left * b[i].right + a[i].right * b[i].left;
  }

  return r;
}
```

This operation is in fact the [dot product] of two vectors containing complex numbers. This function can be rewritten as:

```c
complex float dot(complex *a, complex *b, int n)
{
  complex float r = 0;

  for(int i = 0; i < n; i++)
    r += a[i] * b[i];

  return r;
}
```

[**dot product**] https://en.wikipedia.org/wiki/Dot_product#Coordinate_definition

The functions at 0x11d9 and 0x1284 are used to check for easy solutions.

main

The main function (at 0x1522) is much more complex.

It contains the following parts:

- Generate a vector random numbers, with a random size;
- Read and parse the user's input;
- Perform mathematical operations on the input

Unit vector

The vector r contains n random numbers in [0, 1024[. Its size n is

randomly selected in [3, 6[.

A hint is given to know how big the vector is:

- If n is 3, the program prints Hi!;
- If n is 4, the program prints Hey!;
- If n is 5, the program prints Hoy!.

The vector is dynamically allocated on the stack with alloca.

The vector is then normalized (divided by its norm) and becomes a [unit vector].

```c
// Seed the PRNG with military-grade entropy
srand(time(NULL));

// Pick a random size
int n = 3 + (rand() % 3);
switch(n) {
  case 3: puts("Hi!");  break;
  case 4: puts("Hey!"); break;
  case 5: puts("Hoy!"); break;
}

double *r = alloca(sizeof(*r) * n);

// Fill the vector
for(size_t i = 0; i < n; i++)
  r[i] = rand() % 1024;

// Normalize the vector
double d = norm(n, r);
for(size_t i = 0; i < n; i++)
  r[i] /= d;
```

[unit vector] https://en.wikipedia.org/wiki/Unit_vector

Parse user input

The user input is read on a stack buffer by using scanf. Exploiting this buffer overflow is left as an exercise to the reader.

The input is parsed and stored in a square matrix of dimension n. The elements are read sequentially in the form a$b to represent the complex number

a + bi.

a and b can only be in [0, 15] (with the hexadecimal symbols 0 to F). A negative number can be specified with &

Consider the following input, on a single line, with no spaces:

```
0$1 + 2$3 + 4$5 +
6$7 + 8$9 + A$B +
C$D + E$F + 0$1
```

The 3×3 matrix represented by such input is the following matrix:

```
[ 0 +  1i      2 +  3i     4 +  5i]
[ 6 +  7i      8 +  9i    10 + 11i]
[12 + 13i     14 + 15i     0 +  1i]
```

The matrix is also allocated with alloca. It is possible to smash the r vector to solve this challenge. This was considered a bug, and this write-up does not explain how to exploit this vulnerability. This also means the solution should have 3 different matrices, one for each possible dimension.

Mathematical operations

Ignoring the basic checks on the user's input to make sure the solution is not trivial, the program then performs a few operations on the input matrix.

First, it multiplies the matrix with the unit vector r to a new vector a. Since the content of this vector is random, this hints that the solution either exploits the predictability of the random-number generation algorithm or a relation between the numbers that eventually cancels the vector.

Then, a operation that could not be identified is done on the matrix. It is first [transposed], and every number that is not on the diagonal is replaced by its [conjugate] (i.e. has its imaginary part negated).

This operation is close to the [conjugate transpose], expect for the diagonal.

Consider the following matrix:

```
[ 0 +  1i      2 +  3i     4 +  5i]
[ 6 +  7i      8 +  9i    10 + 11i]
[12 + 13i     14 + 15i     0 +  1i]
```

It becomes:

```
[ 0 +  1i      6 -  7i    12 - 13i]
[ 2 -  3i      8 +  9i    14 - 15i]
[ 4 -  5i     10 - 11i     0 +  1i]
```

This new matrix is then multiplied with the unit vector r again to create b.

[**transposed**] https://en.wikipedia.org/wiki/Transpose

[**conjugate**] https://en.wikipedia.org/wiki/Complex_conjugate

[**conjugate transpose**] https://en.wikipedia.org/wiki/Conjugate_transpose

The vectors a and b are then multiplied to give a complex number s.

The binary accepts inputs when s is 1 + 0i. (i.e. (m × r) × (r × t) = 1)

Finding Solutions

A Sage script was made to reproduce the behaviour of the program. Trying random inputs to get a feel of the algorithm yielded a very interesting result: 1+2i

```
# Randomly generated numbers
r = vector([
        702, 464, 183, # 287, 623
]).normalized()

# Input
l = len(r)

m = matrix(l, l, [
        [1 + 1 * I, 0 + 0 * I, 0 + 1 * I],
        [0 + 1 * I, 1 + 1 * I, 0 + 0 * I],
        [0 + 0 * I, 0 + 1 * I, 1 + 1 * I],
])

# :-(
t = m.transpose()
for i in range(l):
        v = [0] * l
        for j in range(l):
                if i != j:
                        v[j] = t[i][j].conjugate()
                else:
                        v[j] = t[i][j]
        t[i] = v

# Actual maths
a = m * r
b = r * t
s = a * b

print(s)
```

Since the 3×3 case is small enough, it can be solved by a brute-force script that tries every values that make sense in mathematics (-1, 0 and 1)

```
# Randomly generated numbers
r = vector([
        702, 464, 183, # 287, 623
]).normalized()

# Input
l = len(r)

def gen(tries, count, index):
        ret = [tries[0]] * count

        for i in range(count):
                ret[i] = tries[index % len(tries)]
                index //= len(tries)

        return ret


tries = [0, 1, -1]
for loop in range(len(tries) ** (2 * (l * l))):
        if loop == 0:
                continue

        o = loop
        Mi = matrix(l, l, gen(tries, l * l, loop))
        loop //= len(tries) ** (l * l)
        Mr = matrix(l, l, gen(tries, l * l, loop))
        m = Mr + I * Mi

        # :-(
        t = m.transpose()
        for i in range(l):
                v = [0] * l
                for j in range(l):
                        if i != j:
                                v[j] = t[i][j].conjugate()
                        else:
                                v[j] = t[i][j]
                t[i] = v

        a = m * r
        b = r * t
        s = a * b

        if s == 1:
                print(m)
                print(s)
```

```
                    print(o)
```

This script finds the following solution after a few seconds:

```
[0 I 0]
[0 0 I]
[I 0 0]

0$0 + 0$1 + 0$0
0$0 + 0$0 + 0$1
0$1 + 0$0 + 0$0
```

```
% ./qm
Hi!
Now, enter the cavern, give me a password: 0$0+0$1+0$0+0$0+0$0+0$1+0$1+0$0+0$0
Come in!
```

After rewriting a solver in C to get better performances, the following additional solutions were found:

```
[0 0 0 0 I]
[0 0 I 0 0]
[0 0 0 I 0]
[0 I 0 0 0]
[I 0 0 0 0]
0$0+0$0+0$0+0$0+0$1
0$0+0$0+0$1+0$0+0$0
0$0+0$0+0$0+0$1+0$0
0$0+0$1+0$0+0$0+0$0
0$1+0$0+0$0+0$0+0$0
```

```
% ./qm
Hoy!
Now, enter the cavern, give me a password:
0$0+0$0+0$0+0$0+0$1+0$0+0$0+0$1+0$0+0$0+0$0+0$0+0$0+0$1+0$0+0$0+0$1+0$0+0$0+0$0+0$1+
0$0+0$0+0$0+0$0
Come in!
```

```
[0 0 0 I]
[0 0 I 0]
[0 I 0 0]
[I 0 0 0]
0$0+0$0+0$0+0$1
0$0+0$0+0$1+0$0
0$0+0$1+0$0+0$0
0$1+0$0+0$0+0$0
```

```
% ./qm
Hey!
Now, enter the cavern, give me a password:
0$0+0$0+0$0+0$1+0$0+0$0+0$1+0$0+0$0+0$1+0$0+0$0+0$1+0$0+0$0+0$0
Come in!
```

VirtWorld

It appears that there are many possible solutions to this challenge so a keygen script is included in the VirtWorld.zip.

This challenge was a classic, and fun, stack based VM challenge. To solve this we extracted the code from the binary and wrote a “disassembler”. We also attempted to write an emulator which was also provided but it was easiest to solve the challenge statically. The only dynamic modification required was to patch out the move of the ptrace result into a “checked” variable so as to pass the “is_debugged” check and run the program in gdb.

The “pseudo” code for the program looked something like this:

```
goal = 0
BUF[20];

if(ptrace())
    exit(0)

read(1, BUF, 20)

# check 4 bytes at a time
idx = 0
for chunk in range(4, -1, -1):

    char0 = BUF[idx]
    idx += 1
    char1 = BUF[idx]

    if char1 != 95:
        exit(0)

    char0 ^= (95 + chunk)
    idx += 1
    char2 = BUF[idx] - 48 + 14
    idx += 1
    char3 = BUF[idx]
    idx += 1

    # 3rd character affects jmp target
    # 3rd character can either be one of: # 0 1 3 5

    if (char2 + 48 - 14) == ord('#')
        continue
    elif (char2 + 48 - 14) == ord('0')
        if char0 != 0
            exit(0)
    elif (char2 + 48 - 14) == ord('1')
        if char0 + char3 != 0:
            exit(0)
    elif (char2 + 48 - 14) == ord('3')
        if char0 - char3 != 0:
            exit(0)
    elif (char2 + 48 - 14) == ord('5')
        if char0 ^ char3 != 0:
            exit(0)
```

The major optimization in the pseudo code above is the use of the if-else block to abstract away several instructions since the only side effect of the dynamically calculated jmp target was to perform one of 5 checks. The input was checked 4 bytes at a time and the “check” itself was selected based on user input and also not very tight; therefore, many possible combinations were possible for every 4 bytes as long as the check condition held.

For the solution script we assumed we wanted to use check ‘5’ for each 4 byte chunk and simply selected the first solution discovered via brute force.

```
~# ./virtual_world
P_53P_52P_51P_50f_59
GG!
```